Mutagenesis Visualization Documentation

Frank Hidalgo

Jan 31, 2022

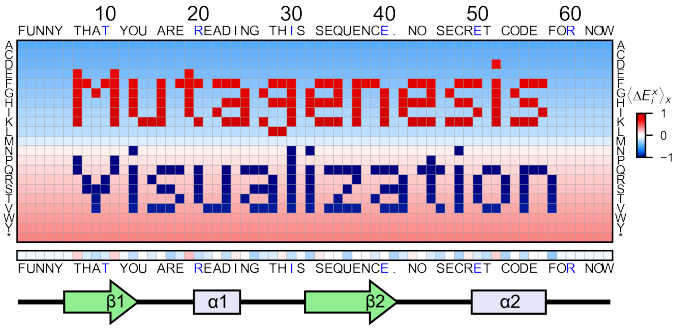
Overview

CHAPTER 1

Overview

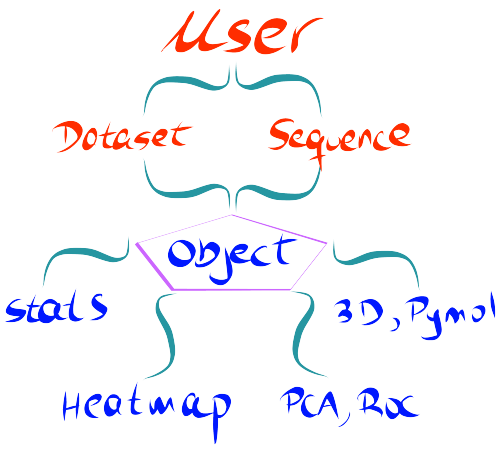
1.1 Description

`mutagenesis_visualization` is a Python package aimed to generate publication-quality figures for saturation mutagenesis datasets.



The package main focus is to perform the statistical analysis and visualization steps of your pipeline, but it additionally offers tools to calculate enrichment scores from fastq files.

Unlike other available python packages, we have developed a user-centered API which does not require prior experience with Python nor statistics. The documentation provides multiple examples of how to perform each step. As the user, you will be guided to input your dataset and the protein sequence. From here, the software *prend le contrôle*, and will produce a wide range of stunning and detailed plots.

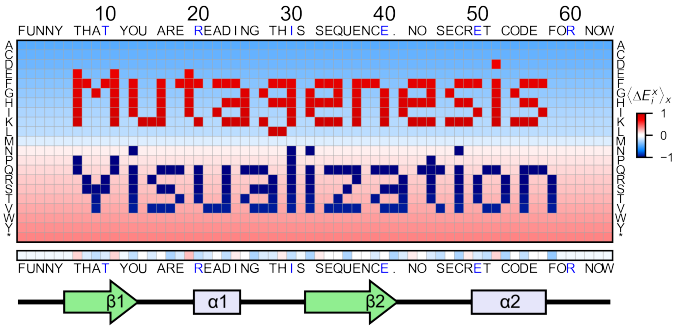


1.2 Key Features

- Calculate enrichment scores from fastq files, allowing for different ways of data processing and normalization.
- Produce publication-quality heatmaps from enrichment scores as well as a wide range of visualization plots.
- Principal component analysis (PCA), hierarchical clustering and receiver operating characteristic (ROC) curve tools.
- Map enrichment scores effortlessly onto a PDB structure using Pymol. Structural properties such as SASA, B-factor or atom coordinates can be extracted from the PDB and visualized using a built-in method.
- Generate dashboards.

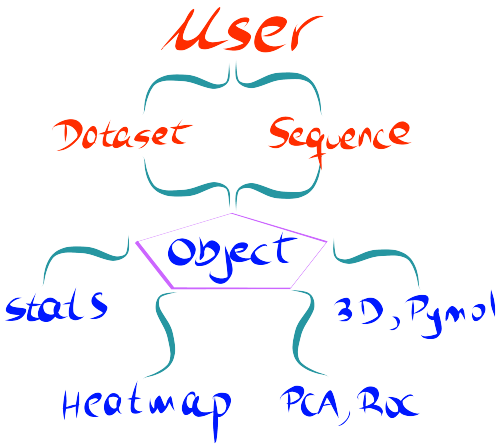
1.3 Description

`mutagenesis_visualization` is a Python package aimed to generate publication-quality figures for saturation mutagenesis datasets.



The package main focus is to perform the statistical analysis and visualization steps of your pipeline, but it additionally offers tools to calculate enrichment scores from FASTQ files.

Unlike other available python packages, we have developed a user-centered API which does not require prior experience with Python nor statistics. The documentation provides multiple examples of how to perform each step. As the user, you will be guided to input your dataset and the protein sequence. From here, the software *prend le contrôle*, and will produce a wide range of stunning and detailed plots.



1.4 Key Features

- Calculate enrichment scores from FASTQ files, allowing for different ways of data processing and normalization.
- Produce publication-quality heatmaps from enrichment scores as well as a wide range of visualization plots.
- Principal component analysis (PCA), hierarchical clustering and receiver operating characteristic (ROC) curve tools.
- Map enrichment scores effortlessly onto a PDB structure using Pymol. Structural properties such as SASA, B-factor or atom coordinates can be extracted from the PDB and visualized using a built-in method.

CHAPTER 2

Getting Started

In this chapter, you will find how to install the package (*Installation guide*) and how to rapidly test that the software is up and running (*Quick demo*). You will also find a workflow.

2.1 Installation guide

2.1.1 Using a virtual environment

The easiest way to run mutagenesis_visualization is by creating a virtual environment, where all the dependencies are installed from scratch. That will avoid errors related to different versions of dependencies. There are different tools to manage virtual environments, such as poetry or conda. We recommend using poetry for its simplicity. A toml, a requirements.txt, and a lock files are present in the repository. Anaconda offers the option to create a virtual environment. Check this guide to know how.

2.1.2 Installing with pip

`mutagenesis_visualization` is compatible with Python =>3.8. The code is available on GitHub under a GNU GENERAL PUBLIC LICENSE. The package can be installed from PyPI using the `pip` package manager by executing the following at the command line:

```
pip install mutagenesis_visualization
```

Note: The package folder is called "mutagenesis_visualization" (with underscore), but pip replaces the underscore with a hyphen. Thus, if you search for it on Pypi, it will show up as "mutagenesis-visualization". For installation purposes, both a hyphen and an underscore work.

2.1.3 Installing from github

Execute the following command to install the library from Github:

```
pip install git+https://github.com/fhidalgor/mutagenesis_visualization
```

2.1.4 Dependencies

In this section I am listing the dependencies and the versions I used to make the package. Check out the requirements.txt and the pyproject.toml files to see the specific versions that were used.

Note: The software has been tested on Linux, Windows and MacOS platforms. The package works in all of them as long as the dependencies are updated. We have encountered issues when the Anaconda environment was old. Those issues got solved by uninstalling and reinstalling Anaconda (which will automatically update all the dependencies.)

Required Dependencies

- python (version >= 3.8)
- numpy
- matplotlib
- seaborn
- pandas
- scipy
- scikit-learn
- biopython
- freesasa
- adjustText
- plotly

- openpyxl
- ipympl
- xlrd
- statsmodels
- xlsxwriter

More updated versions probably work too. In here, we have specified the versions we used when building the software.

If you want to manually install and/or upgrade the dependencies on your own, use:

```
pip install --upgrade package_name
```

Optional dependencies

- ipymol (version 0.5)

`Ipymol` needs to be installed from Github, since the current Pypi version does not work. Make sure that you have a `setuptools` version below 58.0, otherwise there will be an error. To install ipymol, use this command:

```
pip install git+https://github.com/cxhernandez/ipymol
```

You may have already installed `Pymol`. However, if it is not on the same path as Python, there will not be communication between the two. An easy way to circumvent the problem is to reinstall `Pymol` using the following command:

```
conda install -c schrodinger pymol-bundle
```

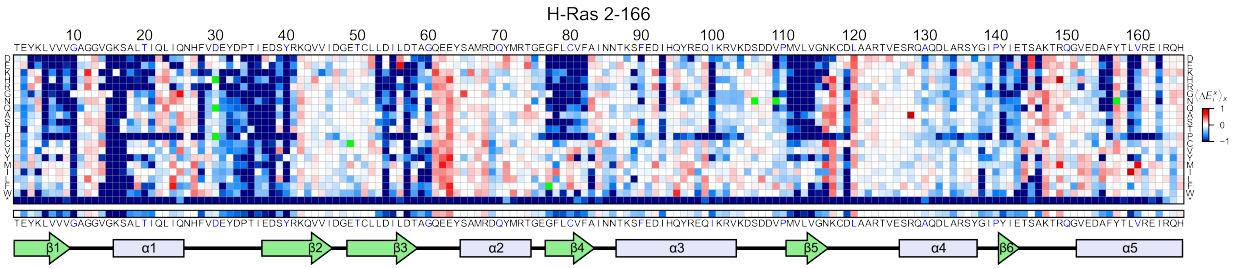
If you create a virtual environment with conda, you need to install the pymol bundle in the virtual environment.

2.2 Quick demo

Now that you have installed `mutagenesis_visualization`, execute the following within Python to evaluate whether it is working propertly:

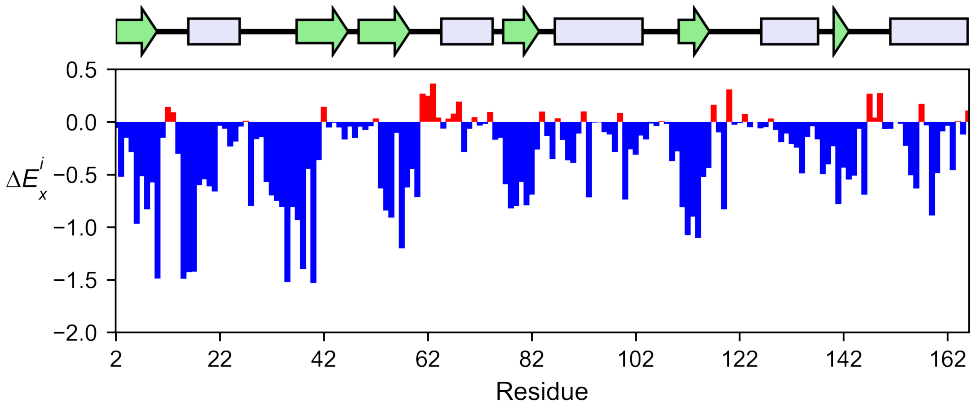
```
import mutagenesis_visualization as mut
mut.run_demo()
```

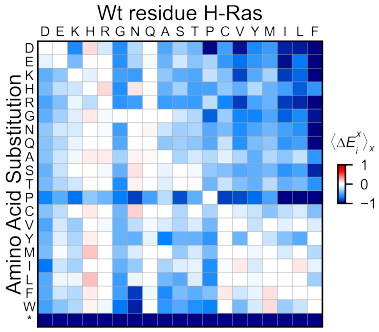
This command will load the `mutagenesis_visualization` package, create a `Screen` object with sample data, call the `object.heatmap` method and show a heatmap plot of the sample data.

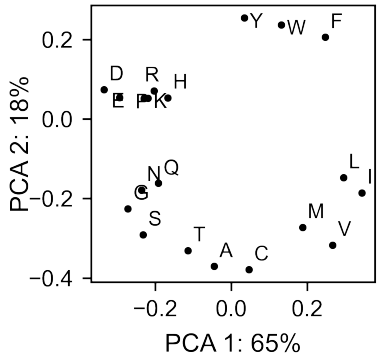


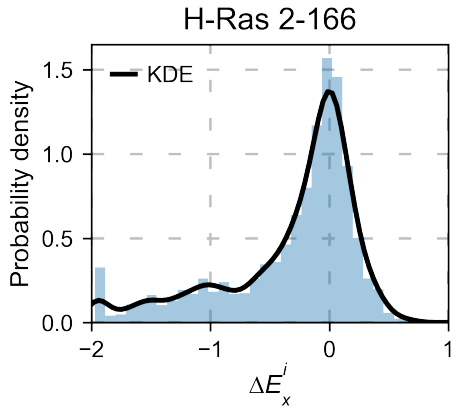
There are four other demo plots, and can be invoked using the following command:

```
mut.run_demo(figure = 'mean')
mut.run_demo(figure = 'miniheatmap')
mut.run_demo(figure = 'kernel')
mut.run_demo(figure = 'pca')
```









Run *mut.run_demo(figure = 'pymol')* to test if your Pymol is connected to this package.

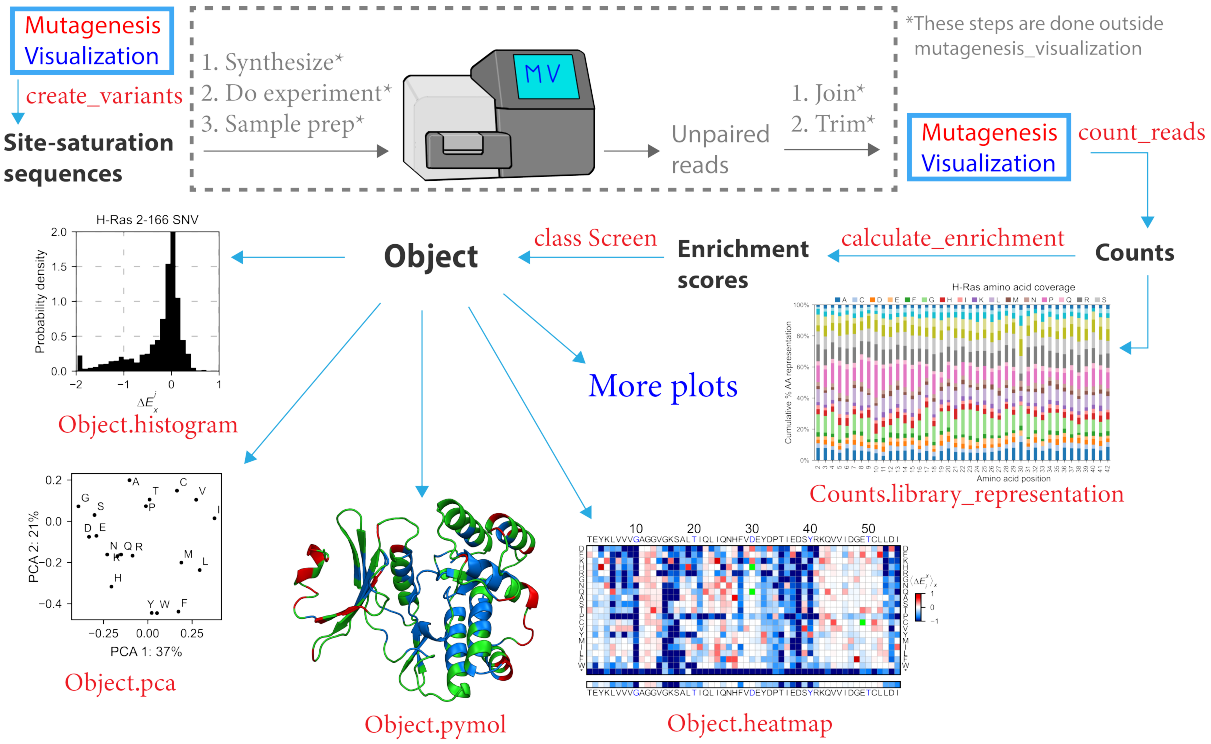
If you would like to play with the data yourself, execute the following command to retrieve the raw data:

```
datasets = mut.load_demo_datasets()
```

A more detailed explanation on how to generate these figures can be seen at *Creating plots* and at *Other datasets*.

2.3 Workflow

Let's take a look to the workflow of this software:



Mutagenesis_visualization will simplify the process of developing and analyzing mutagenesis experiments. To start, you can use this software to design site-saturation sequences using the create_variants function. From here, you will pause your work with Mutagenesis_visualization to synthesize the site-saturation sequences using Twist Bio, Agilent, etc. Once you have got your DNA library ready, you will perform the necessary experiments and sequence the samples. After that, you will use a bioinformatics software (ie Flash) to pair the unpaired reads. Then you will trim the adapters to generate FASTQ files.

Now you will return to the software to conduct analysis of your experiment. Mutagenesis_visualization will read the FASTQ files and return the counts per variant. At this point, there are a few visualization plots that you can create in order to assess the quality of the DNA library. After that, you will calculate the enrichment scores using the calculate_enrichment function (you will need a pre-selection and a post-selection dataset). With the enrichment scores in hand, you can use the Screen class to generate several different plots, including heatmaps, histograms, scatter plots, PCA analysis, Pymol figures, and more.

CHAPTER 3

API Description

In here, you will find the *Classes*, methods and *Functions* used in this API.

3.1 Classes

3.1.1 *CreateVariants* class

class mutagenesis_visualization.**CreateVariants**

Class to create variants for DNA synthesis.

__call__(*dna: str, codon_list: Union[list, str]*) → pandas.core.frame.DataFrame

Generate a list of all point mutants given a dna sequence and a list of codons.

Parameters

- **dna** (*str,*) – Contains the DNA sequence of the allele of reference (usually wild-type).
- **codon_list** (*list or str*) – Input a list of the codons that were used to create point mutations. Example: ["GCC", "GCG", "TGC"]. It is important to know that the order of the codon_list will determine the output order.

Returns **df_output** – Dataframe containing the generated sequences.

Return type pandas dataframe

3.1.2 *Counts* class

class mutagenesis_visualization.**Counts**(*dataframes: Union[pandas.core.frame.DataFrame, List[pandas.core.frame.DataFrame]], start_position: Optional[int] = None, aminoacids: Optional[List[str]] = None*)

Counts represents the output of reading a fastq file.

Parameters

- **dataframes** (*dataframe, list dataframes*) – 2D matrix containing the counts per codon. Columns will contain the amino acid substitutions, rows will contain the counts for each residue in the protein sequence. If multiple replicates, pass items in a list.
- **start_position** (*int, default None*) – First position in the protein sequence that will be used for the first column of the array. If a protein has been mutated only from residue 100-150, then if start_position = 100, the algorithm will trim the first 99 amino acids in the input sequence. The last residue will be calculated based on the length of the input array. We have set the default value to 2 because normally the Methionine in position 1 is not mutated.
- **aminoacids** (*list, default None*) – List of aminoacids (in order). Stop codon needs to be '*'. If none, it will use the index of the dataframe

mean_counts()

library_representation()

LibraryRepresentation pyplot class

class mutagenesis_visualization.main.bar_graphs.library_representation.**Libra**

Class to generate a library representation bar plot.

__call__(*replicate: int = -1, output_file: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generates a cumulative stacked bar plot. Each bar represents an amino acid position,

and each color indicates the observed variant frequency.

Parameters

- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

MeanCounts pyplot class

class mutagenesis_visualization.main.bar_graphs.mean_counts.**MeanCounts**(*datafr*
List[p
po-
si-
tions:
List[in
amino
List[st

Class to generate a mean counts bar plot.

__call__(*normalize: bool = False*, *replicate: int = -1*, *output_file: Union[None*, *str*, *pathlib.Path] = None*, ***kwargs*) → None

Plot in a bargraph the mean counts for each residue of the protein.

Parameters

- **normalize** (*bool, default False*) – If set to true, the mean counts will be normalized so the highest value is 100.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  text_labels [list of lists, default empty] If you want to add a label to the graph, add the coordinates and the text. Example: text_labels = [[x0,y0,text0] [x1,y1,text1]].

3.1.3 *GeneratePrimers* class

class mutagenesis_visualization.**GeneratePrimers**(*dna: str*, *start: str*, *end: str*)

Class that will generate primers for saturation mutagenesis.

__call__(*codon: str = 'NNS'*, *length_primer: int = 15*, *melting_temp: Optional[float] = None*) → pandas.core.frame.DataFrame

Generate primers for saturation mutagenesis.

Parameters

- **codon** (*str, default 'NNS'*) – Degenerate codon that will be used to create the primers. Check idt's website for a list of all mixed bases and letter code (https://www.idtdna.com/pages/products/custom-dna-rna/mixed-bases). This parameter should contain 3 letters, although can contain more.
- **length_primer** (*int, default 15*) – Number of bases that the primers will have to each side of the mutated codon. Total primer length will be 2*length_primer+3.
- **melting_temp** (*int, default None*) – Melting temperature in Celsius of the primers. Will override length_primer. If none, primers will have a total length of 2*length_primer+3

Returns **df** – Dataframe containing the primers.

Return type pandas dataframe

3.1.4 *Screen* class

class mutagenesis_visualization.**Screen**(*datasets: Union[numpy.ndarray[Any, numpy.dtype[ScalarType]], pandas.core.frame.DataFrame, List[Union[numpy.ndarray[Any, numpy.dtype[ScalarType]], pandas.core.frame.DataFrame]]], sequence: str, aminoacids: List[str], start_position: int = 2, delete_position: Union[int, List[int], None] = None, fillna: float = 0, secondary: Optional[List[List[str]]] = None*)

Screen represents a mutagenesis experiment. If you are doing deep scan mutagenesis, then every amino acid in the protein has been mutated to every possible amino acid. For example, if there was a leucine at position 2, then this leucine would be mutated to the other 19

naturally occurring amino acids. However, you can also use the package if you only have a handful of amino acid substitutions.

Parameters

- **datasets** (*array, list arrays, dataframe, list dataframes*) – 2D matrix containing the enrichment scores of the point mutants. Columns will contain the amino acid substitutions, rows will contain the enrichment for each residue in the protein sequence. If multiple replicates, pass items in a list.
- **sequence** (*str*) – Protein sequence in 1 letter code format.
- **aminoacids** (*list, default list('ACDEFGHIKLMNPQRSTVWY*')*) – Amino acid substitutions (rows). Submit in the same order that is used for the array.
- **start_position** (*int, default 2*) – First position in the protein sequence that will be used for the first column of the array. If a protein has been mutated only from residue 100-150, then if start_position = 100, the algorithm will trim the first 99 amino acids in the input sequence. The last residue will be calculated based on the length of the input array. We have set the default value to 2 because normally the Methionine in position 1 is not mutated.
- **delete_position** (*int, List[int], default None*) – Can delete positions (columns) in the dataset. For example, if you set start_position = 2 and delete_position = 122, you will be deleting the column 120 of the input dataset. The sequence parameter won't delete anything, so if you plan on deleting a few columns in your dataset, adjust the input sequence and secondary list.
- **fillna** (*float, default 0*) – How to replace NaN values.
- **secondary** (*list, optional*) – This parameter is used to group the data by secondary structure. The format is the name of the secondary structure multiplied by the residue length of that motif. Example : [['β1']*(8),['L1']*(7),['α1']*(9),...,].
- **replicates** (*list[np.array, Dataframe], optional*) – If you have multiple replicates for that experiment, pass them in the same format as dataset.

dataframe

Contains the enrichment scores, position, sequence.

Type pandas dataframe

Other attributes are same as input parameters

Type dataset, aminoacids,

start_position, roc_df, secondary

The following classes are integrated into *Screen*, thus, you only have to use the __call__ method.

EnrichmentBar pyplot class

class mutagenesis_visualization.main.bar_graphs.enrichment_bar.**EnrichmentBa**

Class to generate a enrichment bar plot per position.

__call__(*mode: str = 'mean'*, *show_cartoon: bool = False*, *min_score: Optional[float] = None*, *max_score: Optional[float] = None*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Plot in a bargraph the enrichment for each residue of the protein. Red for gain of function, blue for loss of function.

Parameters

- **mode** (*str, default 'mean'*) – Specify what enrichment scores to show. If mode = 'mean', it will show the mean of each position. If mode = 'A', it will show the alanine substitution profile. Can be used for each amino acid. Use the one-letter code and upper case.
- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **show_cartoon** (*boolean, default False*) – If true, the plot will display a cartoon with the secondary structure. The user must have added the secondary structure to the object.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  color_gof [str, default 'red'] Choose color to color positions with an enrichment score > 0.

  color_lof [str, default 'blue'] Choose color to color positions with an enrichment score < 0.

Differential pyplot class

class mutagenesis_visualization.main.bar_graphs.differential.**Differential**(*da*
O
tic
=
N
da
O
tic
=
N
ar
U
Li
=
”,
da
O
tic
nu
=
N
se
qu
st
=
”,
se
qu
st
=
”,
st
in
=
0,
se
or
O
tic
=
N
se
or
O
tic
=
N

Class to generate the difference between two experiments.

__call__(*screen_object: Screen*, *metric: Literal[rmse, mean, squared, hard_cutoff] = 'rmse'*, *plot_type: str = 'bar'*, *show_cartoon: bool = False*, *min_score: Optional[float] = None*, *max_score: Optional[float] = None*, *hard_cutoff: Optional[float] = None*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Plot the mean positional difference between two experiments.

Parameters

- **screen_object** (another *Screen* object to compare with.) –
- **metric** (*str, default 'rmse'*) – The way to compare the two objects. Options are 'rmse' ((x-y)**2/N)**0.5, 'squared' ((x**2-y**2)/N, 'mean' (x-y)/N and 'hard_cutoff'. For 'hard_cutoff', select a threshold and the algorithm will count how many mutations are above/below in a pairwise comparison.
- **plot_type** (*str, default 'bar'*) – Options are 'bar' and 'line'.
- **show_cartoon** (*boolean, default False*) – If true, the plot will display a cartoon with the secondary structure. The user must have added the secondary structure to the object.
- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **hard_cutoff** (*float, default None*) – Only works if metric is selected first.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

PositionBar pyplot class

class mutagenesis_visualization.main.bar_graphs.position_bar.**PositionBar**(*dat*
Op-
tion
=
Nor
dat
Op-
tion
=
Nor
am
Uni
Lis
=
”,
dat
Op-
tion
num
=
Nor
se-
que
str
=
”,
se-
que
str
=
”,
star
int
=
0,
sec-
ona
Op-
tion
=
Nor
sec-
ona
Op-
tion
=
Nor

Class to generate a mean enrichment bar plot.

__call__(*position: int*, *mask_selfsubstitutions: bool = False*, *replicate: int = -1*, *output_file: Union[None*, *str*, *pathlib.Path] = None*, ***kwargs*) → None

Choose a position and plot in a bargraph the enrichment score for each substitution. Red for gain of function, blue for loss of function.

Parameters

- **position** (*int*) – number of residue of the protein to display.
- **mask_selfsubstitutions** (*bool, default False*) – If set to true, will assing a score of 0 to each self-substitution. ie (A2A = 0)
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  neworder_aminoacids: list, default list('DEKHRGNQASTPCVYMILFW*')
  Set the order (left to right) of the amino acids.

  color_gof [str, default 'red'] Color to color mutations > 0.

  color_lof [str, default 'blue'] Color to color mutations < 0.

Secondary pyplot class

class mutagenesis_visualization.main.bar_graphs.secondary.**Secondary**(*dataframes: Optional[mute = None*, *dataframes: Optional[List[= None*, *aminoacids: Union[str, List[str]] = ''*, *datasets: Optional[List[numpy.dtyp = None*, *sequence: str = ''*, *sequence_rav str = ''*, *start_posit int = 0*, *secondary: Optional[List[= None*, *secondary_du Optional[List[= None*)

Class to generate bar plot of data sorted by secondary elements.

__call__(*min_score: Optional[float] = None*, *max_score: Optional[float] = None*, *replicate: int = -1*, *show_error_bars: Optional[bool] = True*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generates a bar plot of data sorted by secondary elements (alpha helices and beta sheets).

Parameters

- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **show_error_bars** (*bool, default True*) – If set to true, show error bars measured as the standard deviation of all replicates.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

Kernel pyplot class

class mutagenesis_visualization.main.kernel.kernel.**Kernel**(*dataframes: Optional[mutagenesis_visual = None*, *dataframes_raw: Optional[List[pandas.core.fr = None*, *aminoacids: Union[str, List[str]] = ''*, *datasets: Optional[List[numpy.ndarra numpy.dtype[ScalarType]] = None*, *sequence: str = ''*, *sequence_raw: str = ''*, *start_position: int = 0*, *secondary: Optional[List[T]] = None*, *secondary_dup: Optional[List[T]] = None*)

Class to generate a kernel density plot.

__call__(*show_replicates: bool = False*, *wt_counts_only: bool = False*, *show_mean: bool = False*, *replicate: int = -1*, *output_file: Union[None*, *str*, *pathlib.Path] = None*, ***kwargs*) → None

Plot univariate or bivariate distributions using kernel density estimation.

Parameters

- **show_replicates** (*bool, optional default False*) – If set to true, will plot the kernel of each replicate.
- **wt_counts_only** (*bool, optional default False*) – If set to true, it will plot the kernel distribution of the wild type alleles only.
- **show_mean** (*bool, optional default False*) – If set to true, it will plot the kernel distribution mean of replicates when wt_counts_only is True. Otherwise, it will show the mean by default so this parameter won't work if show_replicates is set to False.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  color: str, default "k" Set the color of the mean plot.

  kernel_color_replicates [list of colors, default None] Add a list of color codes to tune the colors of the plots.

  return_plot_object [boolean, default False] If true, will return plotting objects (ie. fig, ax_object).

Histogram pyplot class

class mutagenesis_visualization.main.kernel.histogram.**Histogram**(*dataframes: Optional[mutagenes = None, dataframes_raw: Optional[List[panda = None, aminoacids: Union[str, List[str]] = '', datasets: Optional[List[nump numpy.dtype[Sca = None, sequence: str = '', sequence_raw: str = '', start_position: int = 0, secondary: Optional[List[T]] = None, secondary_dup: Optional[List[T]] = None*)

Class to generate a histogram plot.

__call__(*population: str = 'All'*, *show_parameters: bool = False*, *loc: str = 'best'*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a histogram plot. Can plot single nucleotide variants (SNVs) or non-SNVs only.

Parameters

- **population** (*str, default 'All'.*) – Other options are 'SNV' and 'nonSNV'.
- **show_parameters** (*bool, default False*) – If set to true, will display the mean and the median of the data.
- **loc** (*str, default "best"*) – Set the location of the meam and median. Check the matplotlib plt.legend method to see how the parameter loc works. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  return_plot_object [boolean, default False] If true, will return plotting objects (ie. fig, ax).

  bins [int or str, default 'auto'.] Number of bins for the histogram. By default it will automatically decide the number of bins.

  color: str, default 'k' Change to a different color if desired.

Sequence pyplot class

class mutagenesis_visualization.main.kernel.sequence_differences.**SequenceDi**

Class to generate the sequence differences plot.

__call__(*screen_object: Screen, map_sequence_changes: List[Tuple[int, int]], legend_labels: Optional[Tuple[str, str]] = None, replicate: int = -1, replicate_second_object: int = -1, output_file: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate two histogram plots. The first plot will have the impact on fitness to go from protein A -> B, and the second plot will contain the B -> A effect.

Parameters

- **screen_object** (*Screen* object or list containing *Screen*) – objects.
- **map_sequence_changes** (*list of tuples*) – Set the residues that differ between protein A and protein B. Example: [(1, 1), (12, 12), (15, 16)]. In the example, the algorithm will compare the residue 1 and 12 of each protein, and the residue 15 of protein A vs the residue 16 of protein B.
- **legend_labels** (*tuple of str*) – Set the labels of the legend.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **replicate_second_object** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  bins [int or str, default 'auto'.] Number of bins for the histogram. By default it will automatically decide the number of bins.

MultipleKernel pyplot class

class mutagenesis_visualization.main.kernel.multiple_kernels.**MultipleKernel**

Class to generate plots of multiple kernels.

Heatmap pyplot class

class mutagenesis_visualization.main.heatmaps.heatmap.**Heatmap**(*dataframes: Optional[mutagenesis_* = *None*, *dataframes_raw: Optional[List[pandas.c* = *None*, *aminoacids: Union[str, List[str]]* = *''*, *datasets: Optional[List[numpy.n numpy.dtype[Scalar* = *None*, *sequence: str* = *''*, *sequence_raw: str* = *''*, *start_position: int* = *0*, *secondary: Optional[List[T]]* = *None*, *secondary_dup: Optional[List[T]]* = *None*)

This class plots a heatmat with the enrichment scores.

__call__(*nancolor: str = 'lime'*, *mask_selfsubstitutions: bool = False*, *color_selfsubstitutions: Optional[str] = 'k'*, *show_cartoon: bool = False*, *show_snv: bool = False*, *hierarchical: bool = False*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a heatmap plot of the enrichment scores.

Parameters

- **nancolor** (*str, default 'lime'*) – Will color np.nan values with the specified color.
- **mask_selfsubstitutions** (*bool, default False*) – If set to true, will assing a score of 0 to each self-substitution. ie (A2A = 0)
- **color_selfsubstitutions** (*str, default black*) – If set to a color, it will color the self-substitution borders. Set to None to not color the self substitutions.
- **show_carton** (*boolean, default False*) – If true, the plot will display a cartoon with the secondary structure. The user must have added the secondary structure to the object.
- **show_snv** (*boolean, default False*) – If true, it will only display mutants that are a single nucleotide variant (SNV) of the wild-type protein sequence. The algorithm does not take into account the wild-type DNA allele, so it will include any possible mutant that is one base away.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  neworder_aminoacids: list, default list('DEKHRGNQASTPCVYMILFW*')
  Order of amino acids (y-axis) to display in the heatmap.

HeatmapColumns pyplot class

class mutagenesis_visualization.main.heatmaps.heatmap_columns.**HeatmapColumns**

This class plots a heatmap with the enrichment scores where you can show selected columns.

__call__(*segment: Tuple[int, int], ylabel: bool = True, nancolor: str = 'lime', mask_selfsubstitutions: bool = False, color_selfsubstitutions: Optional[str] = 'k', replicate: int = -1, output_file: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a heatmap plot enrichment scores but only plots a selected segment.

Parameters

- **segment** (*Tuple[int]*) – Segment is typed as [20,40] and includes both residues 20 and 40.
- **ylabel** (*str, default True*) – Choose False to hide.
- **nancolor** (*str, default 'lime'*) – Will color np.nan values with the specified color.
- **mask_selfsubstitutions** (*bool, default False*) – If set to true, will assing a score of 0 to each self-substitution. ie (A2A = 0)
- **color_selfsubstitutions** (*str, default black*) – If set to a color, it will color the self-substitution borders. Set to None to not color the self substitutions.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

HeatmapRows pyplot class

class mutagenesis_visualization.main.heatmaps.heatmap_rows.**HeatmapRows**(*datafr*
Op-
tional[
=
None,
datafr
Op-
tional[
=
None,
amino
Union
List[st
=
”,
datase
Op-
tional[
numpy
=
None,
se-
quence
str
=
”,
se-
quence
str
=
”,
*start_*
int
=
0,
sec-
ondary
Op-
tional[
=
None,
sec-
ondary
Op-
tional[
=
None)

This class plots a heatmat withe the enrichment scores.

Miniheatmap pyplot class

class mutagenesis_visualization.main.heatmaps.miniheatmap.**Miniheatmap**(*datafra*
Op-
tional[n
=
None,
datafra
Op-
tional[L
=
None,
aminoa
Union[s
List[str]
=
'',
datasets
Op-
tional[L
numpy.a
=
None,
se-
quence:
str
=
'',
se-
*quence_*
str
=
'',
start_po
int
=
0,
sec-
ondary:
Op-
tional[L
=
None,
sec-
*ondary_*
Op-
tional[L
=
None)

Class to generate a ROC analysis.

__call__(*mask_selfsubstitutions: bool = False*, *position_offset: int = 0*, *background_correction: bool = False*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a miniheatmap plot enrichment scores of mutagenesis selection assays.

Parameters

- **mask_selfsubstitutions** (*bool, default False*) – If set to true, will assing a score of 0 to each self-substitution. i.e., (A2A = 0)
- **position_offset** (*int, default 0*) – Will group columns by residues. If the offset is not 0, it will use the values of the n+offset to group by. For example, you may want to see what happens when you have a Proline in front of the mutated residue. The algorithm can report the difference between the calculated value and the mean score for that particular substitution. Offset of 1 means that you evaluate the effect of following residue n+1 on n. Offset of -1 means that you look at the previous residue (n-1 on n).
- **background_correction** (*boolean, default False*) – If offset is nonzero, whether subtract the average effect of a substitution or not.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  colorbar_scale: tuple, default (-1, 1) Scale min and max used in heatmaps and correlation heatmaps.

Rank pyplot class

class mutagenesis_visualization.main.other_stats.rank.**Rank**(*dataframes: Optional[mutagenesis_visu = None, dataframes_raw: Optional[List[pandas.core. = None, aminoacids: Union[str, List[str]] = '', datasets: Optional[List[numpy.ndarr numpy.dtype[ScalarType = None, sequence: str = '', sequence_raw: str = '', start_position: int = 0, secondary: Optional[List[T]] = None, secondary_dup: Optional[List[T]] = None*)

Class to generate a mean enrichment bar plot.

__call__(*mode: str = 'pointmutant'*, *output_file: Union[None*, *str*, *pathlib.Path] = None*, *replicate: int = -1*, ***kwargs*) → None

Generate a rank plot so every mutation/residue is sorted based on enrichment score.

Parameters

- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position
- **outdf** (*boolean, default False*) – If set to true, will return the df with the rank of mutations
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

Cumulative pyplot class

class mutagenesis_visualization.main.other_stats.cumulative.**Cumulative**(*datafr*
Op-
tional[
=
None,
datafr
Op-
tional[
=
None,
amino
Union
List[st
=
”,
datase
Op-
tional[
numpy
=
None,
se-
quence
str
=
”,
se-
quence
str
=
”,
*start_*
int
=
0,
sec-
ondary
Op-
tional[
=
None,
sec-
ondary
Op-
tional[
=
None)

This class will plot a cumulative function on the enrichment scores from first to last amino

acid.

__call__(*mode: str = 'all'*, *replicate: int = -1*, *output_file: Union[None*, *str*, *pathlib.Path] = None*, ***kwargs*) → None

Generates a cumulative plot of the enrichment scores by position.

Parameters

- **mode** (*str, default 'all'*) – Options are 'mean', 'all','SNV' and 'nonSNV'.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

ROC pyplot class

class mutagenesis_visualization.main.other_stats.roc_analysis.**ROC**(*dataframes: Optional[mutage = None, dataframes_ra Optional[List[pa = None, aminoacids: Union[str, List[str]] = '', datasets: Optional[List[nu numpy.dtype[= None, sequence: str = '', sequence_raw: str = '', start_position int = 0, secondary: Optional[List[T] = None, secondary_dup: Optional[List[T] = None*)

Class to generate a ROC analysis.

Correlation pyplot class

class mutagenesis_visualization.main.pca_analysis.correlation.**Correlation**(*da*
O
tic
=
N
da
O
tic
=
N
ar
U
Li
=
”,
da
O
tic
nu
=
N
se
qu
st
=
”,
se
qu
st
=
”,
st
in
=
0,
se
or
O
tic
=
N
se
or
O
tic
=
N

This class will conduct a correlation from the enrichment scores.

__call__(*replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a correlation of each amino acid.

Parameters

- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

  colorbar_scale: tuple, default (-1, 1) Scale min and max used in heatmaps and correlation heatmaps.

IndividualCorrelation pyplot class

class mutagenesis_visualization.main.pca_analysis.individual_correlation.**Ind**

This class will conduct an individual correlation from the enrichment scores.

__call__(*replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generates a bar plot of the correlation of each amino acid mutational profile (row of the heatmap) with the rest of amino acids (rows)

Parameters

- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

PCA pyplot class

class mutagenesis_visualization.main.pca_analysis.pca.**PCA**(*dataframes: Optional[mutagenesis_visual = None*, *dataframes_raw: Optional[List[pandas.core.fr = None*, *aminoacids: Union[str, List[str]] = ''*, *datasets: Optional[List[numpy.ndarra numpy.dtype[ScalarType]] = None*, *sequence: str = ''*, *sequence_raw: str = ''*, *start_position: int = 0*, *secondary: Optional[List[T]] = None*, *secondary_dup: Optional[List[T]] = None*)

This class will conduct a PCA from the enrichment scores.

__call__(*mode: Literal[aminoacid, secondary, residue] = 'aminoacid'*, *dimensions: Tuple[int, int] = (0, 1)*, *adjust_labels: bool = False*, *replicate: int = -1*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generates a plot of two PCA dimensions.

Parameters

- **mode** (*list, default 'aminoacid'*) – Can also do PCA by secondary structure element if set to "secondary" or by individual residue if set to "residue".
- **dimensions** (*tuple, default (0,1)*) – Specify which two PCA dimensions to plot. By default PCA1 vs PCA2. Max dimension is 5.
- **adjust_labels** (*boolean, default False*) – If set to true, it will adjust the text labels so there is no overlap. It is convenient to increase the size of the figure, otherwise the algorithm will not find a solution. Requires to install adjustText package.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) – random_state : int, default 554

DifferentialP plotly class

class mutagenesis_visualization.main.plotly.differential.**DifferentialP**(*datafr*
mu-
ta-
ge-
n-
e-
sis_vis
amino
Union
List[st
=
”,
datase
Op-
tional[
numpy
=
None,
se-
quence
Op-
tional[
=
None,
start_p
Op-
tional[
=
None,
end_p
Op-
tional[
=
None)

This class uses plotly to generate a differential plot.

__call__(*screen_object: Any*, *metric: Literal[rmse, squared, mean] = 'rmse'*, *plot_type: str = 'bar'*, *mode: str = 'mean'*, *replicate: int = -1*, *output_html: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a plotly mean plot.

Parameters

- **screen_object** (another *Screen* object to compare with.) –

- **metric** (*str, default 'rmse'*) – The way to compare the two objects. Options are 'rmse' ((x-y)**2/N)**0.5, 'squared' ((x**2-y**2)/N and 'mean' (x-y)/N.
- **plot_type** (*str, default 'bar'*) – Options are 'bar' and 'line'.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

EnrichmentBarP plotly class

class mutagenesis_visualization.main.plotly.enrichment_bar.**EnrichmentBarP**(*da*
m
ta
ge
n-
e-
si
ar
U
Li
=
”,
da
O
tio
nu
=
N
se
qu
O
tio
=
N
st
O
tio
=
N
er
O
tio
=
N

This class uses plotly to generate a mean enrichment plot.

HeatmapP plotly class

class mutagenesis_visualization.main.plotly.heatmap.**HeatmapP**(*dataframes: mutagenesis_visualization.main aminoacids: Union[str, List[str]] = '', datasets: Optional[List[numpy.nd numpy.dtype[ScalarT = None, sequence: Optional[str] = None, start_position: Optional[int] = None, end_position: Optional[int] = None*)

This class uses plotly to generate a heatmap.

__call__(*mask_selfsubstitutions: bool = False, replicate: int = -1, output_html: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a plotly histogram plot.

Parameters

- **mask_selfsubstitutions** (*bool, default False*) – If set to true, will assing a score of 0 to each self-substitution. ie (A2A =

0)

- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

HistogramP plotly class

class mutagenesis_visualization.main.plotly.histogram.**HistogramP**(*dataframes: mutagenesis_visualization aminoacids: Union[str, List[str]] = '', datasets: Optional[List[num numpy.dtype[Sc = None, sequence: Optional[str] = None, start_position: Optional[int] = None, end_position: Optional[int] = None*)

This class uses plotly to generate a histogram plot.

__call__(*mode: str = 'pointmutant', replicate: int = -1, output_html: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a plotly histogram plot.

Parameters

- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position.

- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

RankP plotly class

class mutagenesis_visualization.main.plotly.rank.**RankP**(*dataframes: mutagenesis_visualization.main.utils.rep aminoacids: Union[str, List[str]] = '', datasets: Optional[List[numpy.ndarray[An numpy.dtype[ScalarType]]]] = None, sequence: Optional[str] = None, start_position: Optional[int] = None, end_position: Optional[int] = None*)

This class uses plotly to generate a rank plot.

__call__(*mode: str = 'pointmutant', replicate: int = -1, output_html: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a plotly rank plot so every mutation/residue is sorted based on enrichment score.

Parameters

- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

Scatter3DPDB plotly class

class mutagenesis_visualization.main.plotly.scatter_3d_pdb.**Scatter3DPDB**(*dataf*
mu-
ta-
ge-
n-
e-
sis_v
amin
Unio
List[
=
”,
datas
Op-
tiona
nump
=
None
se-
quen
Op-
tiona
=
None
*start_*
Op-
tiona
=
None
*end_*
Op-
tiona
=
None

This class uses plotly to generate a 3D scatter plot of the protein and the enrichment scores where you can add PDB properties.

__call__(*pdb_path: str = None*, *plot: Optional[List[str]] = None*, *mode: str = 'mean'*, *custom: Any = None*, *position_correction: int = 0*, *chain: str = 'A'*, *replicate: int = -1*, *output_html: Union[None*, *str*, *pathlib.Path] = None*, ***kwargs*) → None

Generates a 3-D scatter plot of different properties obtained from the PDB. PDBs may have atoms missing, you should fix the PDB before using this method. We recommend

you use matplotlib for interactive plot.

Parameters

- **pdb_path** (*str, default None*) – User should specify the path PDB.
- **plot** (*list, default ['Distance', 'SASA', 'log B-factor']*) – List of 3 elements to plot. Other options are 'Score' and Custom. If custom, add the label to the third element of the list ie. ['Distance', 'SASA', 'Conservation'].
- **mode** (*str, default 'mean'*) – Specify what enrichment scores to use. If mode = 'mean', it will use the mean of each position to classify the residues. If mode = 'A', it will use the Alanine substitution profile. Can be used for each amino acid. Use the one-letter code and upper case.
- **custom** (*list or dataframe or np.array, default None*) – If you want to add a custom dataset to plot, use custom. On the parameter plot, the 3rd item of the list will be the label for your custom dataset.
- **df_color** (*pandas dataframe, default None*) – The color of each residue can also be included. You must label that label column.
- **color_by_score** (*boolean, default True*) – If set to False, the points in the scatter will not be colored based on the enrichment score.
- **position_correction** (*int, default 0*) – If the pdb structure has a different numbering of positions than you dataset, you can correct for that. If your start_position = 2, but in the PDB that same residue is at position 20, position_correction needs to be set at 18.
- **chain** (*str, default 'A'*) – Chain of the PDB file to get the coordinates and SASA from.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

Scatter3D plotly class

class mutagenesis_visualization.main.plotly.scatter_3d.**Scatter3D**(*dataframes: mutagenesis_visualization aminoacids: Union[str, List[str]] = '', datasets: Optional[List[num numpy.dtype[Sc = None, sequence: Optional[str] = None, start_position: Optional[int] = None, end_position: Optional[int] = None*)

This class uses plotly to generate a 3D scatter plot of the protein and the enrichment scores.

__call__(*pdb_path: str, mode: str = 'mean', df_coordinates: bool = None, position_correction: int = 0, chain: str = 'A', squared: bool = False, replicate: int = -1, output_html: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generates a 3-D scatter plot of the x,y,z coordinates of the C-alpha atoms of the residues, color coded by the enrichment scores. PDBs may have atoms missing, you should fix the PDB before using this method. Use matplotlib for interactive plot.

Parameters

- **pdb** (*str, default None*) – User should specify the path PDB.
- **mode** (*str, default 'mean'*) – Specify what enrichment scores to use. If mode = 'mean', it will use the mean of each position to classify the residues. If mode = 'A', it will use the Alanine substitution profile. Can be used for each amino acid. Use the one-letter code and upper case.
- **df_coordinates** (*pandas dataframe, default None*) – If no pdb is included, the user must pass the 3-D coordinates of the residues to plot. In here you have more flexibility and you can select other atoms besides the C-alpha.
- **position_correction** (*int, default 0*) – If the pdb structure has a different numbering of positions than you dataset, you can correct for that. If your start_position = 2, but in the PDB that same residue is at position 20, position_correction needs to be set at 18.
- **chain** (*str, default 'A'*) – Chain of the PDB file to get the coordinates and SASA from.
- **squared** (*boolean, False*) – If this parameter is True, the algorithm will center the data, and plot the square value of the distance.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

ScatterP plotly class

class mutagenesis_visualization.main.plotly.scatter.**ScatterP**(*dataframes: mutagenesis_visualization.main aminoacids: Union[str, List[str]] = '', datasets: Optional[List[numpy.nd numpy.dtype[ScalarT = None, sequence: Optional[str] = None, start_position: Optional[int] = None, end_position: Optional[int] = None*)

This class uses plotly to generate a scatter plot.

__call__(*screen_object: Any, mode: str = 'pointmutant', show_results: bool = False, replicate: int = -1, output_html: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a scatter plot between object and a second object of the same class.

Parameters

- **screen_object** (object from class *Screen* to do the scatter with) –

- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position.
- **show_results** (*boolean, default False*) – If set to true, will export the details of the linear fit.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_html** (*str, default None*) – If you want to export the generated graph into html, add the path and name of the file. Example: 'path/filename.html'.
- ****kwargs** (*other keyword arguments*) –

Scatter pyplot class

class mutagenesis_visualization.main.scatter.scatter.**Scatter**(*dataframes: Optional[mutagenesis_v = None*, *dataframes_raw: Optional[List[pandas.co = None*, *aminoacids: Union[str, List[str]] = ''*, *datasets: Optional[List[numpy.nd numpy.dtype[ScalarT = None*, *sequence: str = ''*, *sequence_raw: str = ''*, *start_position: int = 0*, *secondary: Optional[List[T]] = None*, *secondary_dup: Optional[List[T]] = None*)

Class to generate a kernel density plot.

__call__(*screen_object: Union[Screen, Any], mode: Literal[mean, pointmutant] = 'pointmutant', min_score: Optional[float] = None, max_score: Optional[float] = None, replicate: int = -1, replicate_second_object: int = -1, output_file: Union[None, str, pathlib.Path] = None, **kwargs*) → None

Generate a scatter plot between object and a second object of the same class.

Parameters

- **screen_object** (object from class *Screen* to do the scatter with) –
- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position.
- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **replicate_second_object** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

ScatterReplicates pyplot class

class mutagenesis_visualization.main.scatter.scatter_replicates.**ScatterRepli**

Class to generate scatter plots of each pairwise replicate combination.

__call__(*wt_counts_only: bool = False*, *mode: Literal[mean, pointmutant] = 'pointmutant'*, *min_score: Optional[float] = None*, *max_score: Optional[float] = None*, *output_file: Union[None, str, pathlib.Path] = None*, ***kwargs*) → None

Generate a series of scatter plots between replicates.

Parameters

- **wt_counts_only** (*bool, optional default False*) – If set to true, it will plot the kernel distribution of the wild type alleles only. mode will be pointmutant by default.
- **mode** (*str, default 'pointmutant'.*) – Alternative set to "mean" for the mean of each position.
- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **output_file** (*str, default None*) – If you want to export the generated graph, add the path and name of the file. Example: 'path/filename.png' or 'path/filename.svg'.
- ****kwargs** (*other keyword arguments*) –

Pymol pyplot class

class mutagenesis_visualization.main.pymol.pymol.**Pymol**(*dataframes: Optional[mutagenesis_visualizati = None, dataframes_raw: Optional[List[pandas.core.frame. = None, aminoacids: Union[str, List[str]] = '', datasets: Optional[List[numpy.ndarray[An numpy.dtype[ScalarType]]]] = None, sequence: str = '', sequence_raw: str = '', start_position: int = 0, secondary: Optional[List[T]] = None, secondary_dup: Optional[List[T]] = None*)

This class acts as a wrapper with the ipymol github repo.

__call__(*pdb: Union[str, pathlib.Path], mode: str = 'mean', residues: List[str] = None, position_correction: int = 0, esthetic_parameters: bool = True, min_score: Optional[float] = None, max_score: Optional[float] = None, replicate: int = -1, **kwargs*) → None

Color pymol structure residues. User can specify the residues to color, or can use the mutagenesis data. Activating mutations will be colored red and loss of function

blue. Neutral mutations in green. Only works if pymol is your $PATH as pymol or you can start PyMOL in server mode. Uses the ipymol package, which needs to be installed from Github $pip install git+https://github.com/cxhernandez/ipymol, not from pypi (not updated here).

Please ensure that PyMOL is in your $PATH as pymol.

Parameters

- **pdb** (*str*) – User should specify the PDB chain in the following format 4G0N_A. If you have internet connection, Pymol will download the pdb. Otherwise, include the path were your PDB is stored locally.
- **mode** (*str, default 'mean'*) – Others: 'snv' 'nonsnv', 'aminoacid' Specify what enrichment scores to use. If mode = 'mean', it will use the mean of each position to classify the residues. If mode = 'A', it will use the Alanine substitution profile. Can be used for each amino acid. Use the one-letter code and upper case.
- **residues** (*list , optional*) – If user decides to pass custom arguments, use the following format residues = ['1,2,3,4-10','12-15,23,24,35','48,49,50,52-60'] which are [blue,red,green].
- **position_correction** (*int, default 0*) – If the pdb structure has a different numbering of positions than you dataset, you can correct for that. If your start_position = 2, but in the PDB that same residue is at position 20, position_correction needs to be set at 18.
- **esthetic_parameters** (*bool, default True*) – If set to True, pymol will apply the mutagenesis_visualization custom parameters instead of the default Pymol ones.
- **min_score** (*float, default None*) – Change values below a minimum score to be that score. i.e., setting min_score = -1 will change any value smaller than -1 to -1.
- **max_score** (*float, default None*) – Change values below a maximum score to be that score. i.e., setting max_score = 1 will change any value greater than 1 to 1.
- **replicate** (*int, default -1*) – Set the replicate to plot. By default, the mean is plotted. First replicate start with index 0. If there is only one replicate, then leave this parameter untouched.
- ****kwargs** (*other keyword arguments*) –

  gof [int, default is 1] cutoff for determining gain of function mutations based on mutagenesis data.

  lof [int, default is -1] cutoff for determining loss of function mutations based on mutagenesis data.

color [str, default 'chlorine'] Choose color to color neutral.

color_gof [str, default 'red'] Choose color to color positions with an enrichment score > gof.

color_lof [str, default 'neptunium'] Choose color to color positions with an enrichment score < lof.

Returns

- *Open pymol session with a fetched pdb structure where the residues*
- *are colored according to the enrichment scores.*

3.2 Functions

mutagenesis_visualization.**calculate_enrichment**(*aminoacids: Union[List[str], str], pre_lib: Union[str, pandas.core.frame.DataFrame, numpy.ndarray[Any, numpy.dtype[ScalarType]]], post_lib: Union[str, pandas.core.frame.DataFrame, numpy.ndarray[Any, numpy.dtype[ScalarType]]], pre_wt: Union[str, None, numpy.ndarray[Any, numpy.dtype[ScalarType]]] = None, post_wt: Union[str, None, numpy.ndarray[Any, numpy.dtype[ScalarType]]] = None, zeroing_method: Literal[none, zscore, counts, wt, wt synonymous, kernel, population] = 'population', zeroing_metric: Literal[mean, mode, median] = 'median', stopcodon: bool = True, min_counts: int = 25, min_countswt: int = 100, std_scale: Optional[float] = 0.2, mad_filtering: int = 2, mwt: float = 2, infinite: float = 3, output_file: Union[None, str, pathlib.Path] = None*) → numpy.ndarray[Any, numpy.dtype[ScalarType]]

Determine the enrichment scores of a selection experiment, where there is a preselected

population (input) and a selected population (output).

Parameters

- **aminoacids** (*list, str*) – Index of aminoacids (in order). Stop codon needs to be '*'.
- **pre_lib** (*str, pandas dataframe or np.array*) – Can be filepath and name of the exported txt file, dataframe or np.array.
- **post_lib** (*str, pandas dataframe or np.array*) – Can be filepath and name of the exported txt file, dataframe or np.array.
- **pre_wt** (*str, or np.array, optional*) – Str with filepath and name of the exported txt file or np.array.
- **post_wt** (*str, or np.array, optional*) – Str with filepath and name of the exported txt file or np.array.
- **zeroing_method** (*str, default 'population'*) – Method to normalize the data. Can also use 'none', 'zscore', 'counts', 'wt' or 'kernel'. If 'wt' is used 'pre_wt' must not be set to None.
- **zeroing_metric** (*str, default 'median'*) – Metric to zero the data. Only works if zeroing_method='population' or 'wt'. Can also be set to 'mean' or 'mode'.
- **stopcodon** (*boolean, default False*) – Use the enrichment score stop codons as a metric to determine the minimum enrichment score.
- **min_counts** (*int, default 25*) – If mutant has less than the min_counts, it will be replaced by np.nan.
- **min_countswt** (*int, default 100*) – If synonymous wild-type mutant has less than the min_counts, it will be replaced by np.nan.
- **std_scale** (*float, default 0.2*) – Factor by which the population is scaled. Set to None if you don't want to scale the data.
- **mad_filtering** (*int, default 2*) – Will apply MAD (median absolute deviation) filtering to data.
- **mwt** (*int, default 2*) – When MAD filtering is applied, mad_filtering is the number of medians away a data point must be to be discarded. mwt is only used when the population of wild-type alleles is the reference for data zeroing_method.
- **infinite** (*int, default 3*) – It will replace +infinite values with +3 and -infinite with -3.

- **output_file** (*str, default None*) – If you want to export the generated files, add the path and name. Example: 'path/filename.txt'. File will be save as a txt, csv, xlsx file.

Returns zeroed – A np.array containing the enrichment scores.

Return type ndarray

mutagenesis_visualization.**count_reads**(*dna_sequence: str, input_file: Union[str, pathlib.Path], codon_list: Union[List[str], str] = 'NNS', counts_wt: bool = True, start_position: int = 2, output_file: Union[None, str, pathlib.Path] = None, full: bool = False*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]

Process a trimmed fastq file containing DNA reads and returns the counts of each DNA sequence specified by the user.

Parameters

- **dna_sequence** (*str,*) – Contains the DNA sequence of the allele of reference (usually wild-type).
- **input_file** (*str, default None*) – Path and name of the fastq file (full name including suffix ".fastq").
- **codon_list** (*list or str, default 'NNS'*) – Input a list of the codons that were used to create point mutations. Example: ["GCC", "GCG", "TGC"]. If the library was built using NNS and NNK codons, it is enough to input 'NNS' or 'NNK' as a string. It is important to know that the order of the codon_list will determine the output order.
- **counts_wt** (*boolean, default True*) – If true it will add the counts to the wt allele. If false, it will set it up to np.nan.
- **start_position** (*int, default 2*) – First position in the protein sequence that will be used for the first column of the array. If a protein has been mutated only from residue 100-150, then if start_position = 100, the algorithm will trim the first 99 amino acids in the input sequence. The last residue will be calculated based on the length of the input array. We have set the default value to 2 because normally the Methionine in position 1 is not mutated.
- **output_file** (*str, default None*) – If you want to export the generated files, add the path and name of the file without suffix. Example: 'path/filename.xlsx'.
- **full** (*bool, optional*) – Switch determining nature of return

value. When it is False (the default) just the reads are returned, when True diagnostic information from the fastq analysis is also returned.

Returns

- **df_counts** (*dataframe*) – Dataframe with the counts for each point mutant.
- **wt_counts** (*list*) – List of the counts for each for each DNA sequence that codes for the wild-type protein.
- **useful_reads** (*str*) – Present only if *full* = True. Contains the useful reads.

mutagenesis_visualization.**count_fastq**(*variants: List[str], input_file: Union[str, pathlib.Path]*) → Tuple[dict, int, int]

Count the frequency of variants in the input fastq file.

Parameters

- **variants** (*list*) –
- **input_file** (*str, default None*) – Path and name of the fastq file (full name including suffix ".fastq").

Returns

- **variants** (*ordered dict*) – Same input dictionary by now has the values updated with the counts.
- **totalreads** (*int*) – Total number of DNA chains that appear in the fastq file.
- **usefulreads** (*int*) – Total number of identified DNA chains. Calculated as the sum of all the key values.

mutagenesis_visualization.**load_demo_datasets**() → Dict[str, pandas.core.frame.DataFrame]

Loads example datasets so the user can play with it.

Returns **data_dict** – Dictionary that contains the datasets used to create the plots on the documentation.

Return type Dict[str, DataFrame]

mutagenesis_visualization.**run_demo**(*figure: Literal[heatmap, miniheatmap, mean, kernel, pca, position, secondary_mean, correlation, individual_correlation, pymol] = 'heatmap', show: bool = True*) → None

Performs a demonstration of the mutagenesis_visualization software.

Parameters

- **figure** (*str, default 'heatmap'*) – There are a few example plots that can be displayed to test the package is working on your station. The options are 'heatmap', 'miniheatmap', 'mean', 'kernel', 'pca' 'position', 'secondary_mean', 'correlation', 'individual_correlation' and 'pymol'. Check the documentation for more information.
- **show** (*boolean, default True*) – If True, will execute plt.show() for each figure.

The following function *generate_default_kwargs* is not called by the user as a function. It contains the kwargs that are parameters of the Screen methods.

mutagenesis_visualization.main.utils.kwargs.**generate_default_kwargs**() → Dict[str, Any]

Kwargs used in the methods and some other functions. Not all kwargs work on each method, read the individual description. Don't call this function on its own, use the parameters within the plotting methods.

Example: mut.heatmap(colormap=colormap of interest)

Parameters

- **colormap** (*cmap, default custom bluewhitered*) – Used for heatmaps. You can use your own colormap or the ones provided by matplotlib. Example colormap = copy.copy((plt.cm.get_cmap('Blues_r')))
- **colorbar_scale** (*tuple, default [-1, 1]*) – Scale min and max used in heatmaps and correlation heatmaps.
- **color** (*str, default 'k'*) – Color used for the kernel plot line, the histogram, the bar plots.
- **title** (*str, default 'Title'*) – Title of plot.
- **x_label** (*str, default 'x_label'*) – Label of x axis.
- **y_label** (*str, default 'y_label'*) – Label of y axis.
- **xscale** (*tuple, default (None, None)*) – MinMax of x axis.
- **yscale** (*tuple, default (None, None)*) – MinMax of y axis.
- **tick_spacing** (*int, default 1*) – Space of axis ticks. Used for scatter and cumulative plots.
- **outputfilepath** (*str, default ''*) – Path where file will be exported to.
- **outputfilename** (*str, default ''*) – Name of the exported file.

- **dpi** (*int, default 600*) – Dots Per Inch in the created image.
- **neworder_aminoacids** (*list, default list('DEKHRGNQASTPCVYMILFW*')*) – Order of amino acids to display in heatmaps. Used for heatmaps.
- **gof** (*int, default 1*) – Cutoff of the enrichment score to classify a mutation as gain of function. Used on pymol and 3D methods.
- **lof** (*int, default -1*) – Cutoff of the enrichment score to classify a mutation as loss of funtion. Used on pymol and 3D methods.
- **color_gof** (*str, default 'red'*) – Color to color mutations above the gof cutoff. Used in pymol, 3D and mean methods.
- **color_lof** (*str, default 'blue'*) – Color to color mutations below the lof cutoff. Used in pymol, 3D and mean methods.
- **cartoon_colors** (*list, default ['lightgreen', 'lavender', 'k']*) – Colors used for secondary structure cartoon. Used for heatmap, mean and mean_count plots.
- **text_labels** (*str, default 'None'*) – Text labels that you can add to mean and mean_count plots. You will need to specify the coordinates.
- **show** (*boolean, default True*) – Whether to execute plt.show() or not on a matplotlib object.
- **close** (*boolean, default False*) – Whether to execute plt.close() or not on a matplotlib object.
- **random_state** (*int, default 554*) – Random state used for PCA function.
- **bins** (*int or str, default 'auto'.*) – Number of bins for the histogram. By default it will automatically decide the number of bins.
- **return_plot_object** (*boolean, default False*) – If true, will return plotting object.
- **figsize_x** (*int*) –
- **figsize_y** (*int*) –
- **legend_fontsize** (*int, default 10*) –

Returns **default_kwargs** – Dictionary with the default kwargs.

Return type dict

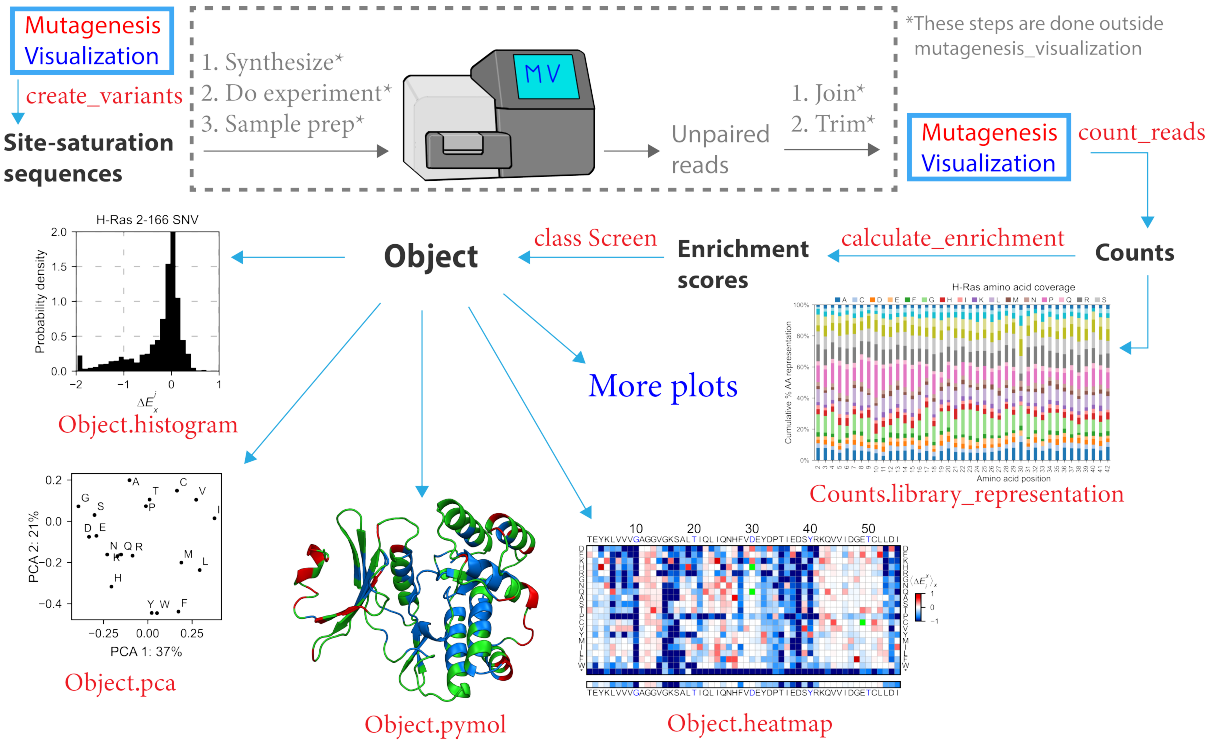
CHAPTER 4

Tutorial

In this chapter, we will walk the user through the different functions and methods of this Python library. You can access to the tutorial via mybinder . We will start with *Design DNA libraries* by seeing how to generate the primers to synthesize the DNA library, or the input FASTA file containing all possible site-saturation sequences that companies like Twist Bioscience need in order to synthesize the library for you. Then, from a FASTQ file, we will process the data (*Processing DNA reads*) and we will do each type of plot (*Creating heatmaps* and *Creating plots*). *Normalizing datasets* shows the different options of data normalization that the package allows for. *Other datasets* uses other datasets to showcase the different options that the software gives you. The jupyter notebooks used to generate the examples can be found on GitHub and are named `doc1_library.ipynb`, `doc2_processing.ipynb`, `doc3_normalizing.ipynb`, `doc4a_plotting_heatmaps.ipynb`, `doc4b_plotting_stats.ipynb`, `doc5_plotly.ipynb` and `doc6_other_datasets.ipynb`.

4.1 Tutorial introduction

Let's take a look to the workflow:



To start, you can use this software to **design site-saturation sequences** (*doc1_library.ipynb*). From here, you will pause your work with Mutagenesis_visualization to synthesize the site-saturation sequences using Twist Bio, Agilent, etc. Once you have got your DNA library ready, you will perform the necessary experiments and sequence the samples. After that, you will use a bioinformatics software (ie Flash) to pair the unpaired reads. Then you will trim the adapters to generate FASTQ files.

Now you will return to the software to conduct the **data processing** of your experiment (*doc2_processing.ipynb*). Mutagenesis_visualization will read the FASTQ files and return the counts per variant. At this point, there are a few visualization plots that you can create in order to assess the quality of the DNA library. After that, you will calculate the enrichment scores using the calculate_enrichment function (you will need a pre-selection and a post-selection dataset). There are different ways of conducting the **data normalization**, and you should see what parameters fit your interests best (*doc3_normalizing.ipynb*).

With the enrichment scores in hand, you will have multiple options to **plot and visualize the data**, including heatmaps, histograms, scatter plots, PCA analysis, Pymol figures, and more (*doc4a_heatmaps.ipynb* and *doc4b_plotting.ipynb*) and (*doc5_plotly.ipynb*). We have compiled **other people's datasets** and visualized them (*doc6_other_datasets.ipynb*).

You can access the jupyter notebooks and play with the code in mybinder.

4.2 Design DNA libraries

In this section we will generate the primers that are used to do saturation mutagenesis on proteins (ie. NNS primers).

We will also generate each possible point mutant sequence and export it to a Fasta file, which can be useful if you use Twist Bioscience to generate your site saturation library.

Classes reviewed in this notebook:

- *mutagenesis_visualization.GeneratePrimers*
- *mutagenesis_visualization.CreateVariants*

4.2.1 Design primers

Now we will define the dna sequence, the beginning and end of the mutable part.

```
from typing import List
from pandas.core.frame import DataFrame
from mutagenesis_visualization import GeneratePrimers

# DNA
dna: str =
 ↪'TGTACAGTAATACAAGGGGTGTTATGGAAAAAATTATGCCGGAAGAAGAATACAGCGAATTTAAAGAACTGATTCTGCAC
 ↪'
                                                  #^
# Start of protein. Note 'ATG' codon is the first codon.
start: str = 'ATGGAAAAAATTATGCCGGAAGAA'

# The 'tat' codon will be the first codon that is not mutated
end: str = 'tatccgtatgatgtgccggattatgcg'

# Initialize instance of class GeneratePrimers
generate_primers : GeneratePrimers = GeneratePrimers(dna, start, end)
```

Set all primers to have the same base pair length.

```
df_primers: DataFrame = generate_primers(codon='NNS', length_primer=15)
```

Set all primers to have the same melting temperature.

```
df_primers_tm: DataFrame = generate_primers(codon='NNS', melting_
 ↪temp=60)
```

If you just want to export the file to excel. This command must be run after first generating a dataframe.

```
generate_primers.export_file(output_file="path/to/file.xlsx")
```

FP_label	FP_seq	RP_label	RP_seq
	CAGTAATACAAGGGGTGTTNNS		CCGGCATAATTTTTTCSNNAACAC
fp 0	GAAAAAATTATGCCGG	rp 0	CCCTTGTATTACTG
	GTAATACAAGGGGTGTTATGN		TCTTCCGGCATAATTTTSNNCATA
fp 1	NSAAAATTATGCCGGAAGA	rp 1	ACACCCCTTGTATTAC
	CAAGGGGTGTTATGGAANNSA		TCTTCCGGCATAATSNNTTCCATA
fp 2	TTATGCCGGAAGA	rp 2	ACACCCCTTG
	GGGGTGTTATGGAAAAANNSA		TCTTCTTCCGGCATSNNTTTTTCCA
fp 3	TGCCGGAAGAAGA	rp 3	TAACACCCC
	GGGGTGTTATGGAAAAAATTN		CTGTATTCTTCTTCCGGSNNAATT
fp 4	NSCCGGAAGAAGAATACAG	rp 4	TTTTCCATAACACCCC
	GGGTGTTATGGAAAAAATTATG		ATTCGCTGTATTCTTCTTCSNNCAT
fp 5	NNSGAAGAAGAATACAGCGAA	rp 5	AATTTTTTCCATAACACCC
	GTTATGGAAAAAATTATGCCGN		AAATTCGCTGTATTCTTCSNNCGG
fp 6	NSGAAGAATACAGCGAATTT	rp 6	CATAATTTTTTCCATAAC

4.2.2 Design site-saturation sequences

Define dna sequence and the list of codons that we want to use to generate the mutants.

```
from mutagenesis_visualization import CreateVariants

# list of codons we want to use
codon_list: List[str] = ["GCC", "GCG", "TGC", "GAC", "GAG", "TTC"]
# DNA sequence we are going to use as the template
dna: str =
 ↪'ATGGCCGTGGGGTGTTATGGATGTACAGTAATACAAGGGGTGTTATGGAAAAAATTATGCCGGAAGAAGAATACAGCGAA
 ↪'

# Initialize instance
create_variants: CreateVariants = CreateVariants()
```

Get a dataframe with the sequences:

```
df_variants: DataFrame = create_variants(dna, codon_list)
```

If you just want to export the file to fasta. This command must be run after first generating a dataframe.

```
create_variants.export_file(output_file="path/to/sequences.fasta")
```

```
>0
ATGGCCGTGGGGTGTTATGGATGTACAGTAATAC
AAGGGGTGTTATGGAAAAAATTATGCCGGAAGAA
GAATACAGCGAATTTAAAG
>1
GCCGCCGTGGGGTGTTATGGATGTACAGTAATAC
AAGGGGTGTTATGGAAAAAATTATGCCGGAAGAA
GAATACAGCGAATTTAAAG
>2
GCGGCCGTGGGGTGTTATGGATGTACAGTAATAC
AAGGGGTGTTATGGAAAAAATTATGCCGGAAGAA
GAATACAGCGAATTTAAAG
>3
TGCGCCGTGGGGTGTTATGGATGTACAGTAATAC
AAGGGGTGTTATGGAAAAAATTATGCCGGAAGAA
GAATACAGCGAATTTAAAG
>4
GACGCCGTGGGGTGTTATGGATGTACAGTAATAC
AAGGGGTGTTATGGAAAAAATTATGCCGGAAGAA
GAATACAGCGAATTTAAAG
```

If you just want to export the file to excel:

```
create_variants.export_file(output_file="path/to/sequences.xlsx")
```

4.3 Processing DNA reads

This section will teach you how to use the built-in data muting functions. If you already have your own muting pipeline built, you can skip this section and go to the plotting examples.

4.3.1 Import module

```
%matplotlib inline
from typing import List
import numpy as np
import pandas as pd
from pandas.core.frame import DataFrame
from mutagenesis_visualization import count_reads
from mutagenesis_visualization import count_fastq
from mutagenesis_visualization import calculate_enrichment

from mutagenesis_visualization.main.utils.data_paths import HRAS_FASTQ,
↪ HRAS_GAPGEF_COUNTS
from mutagenesis_visualization import Counts
from mutagenesis_visualization import Screen
```

4.3.2 Count DNA reads from fastq file

Site saturation mutagenesis

Methods and functions reviewed in this notebook:

- *mutagenesis_visualization.count_fastq()*
- *mutagenesis_visualization.count_reads()*
- *mutagenesis_visualization.Counts*

After sequencing your DNA library, using other packages you will assemble the forward and reverse reads and trim the flanking bases. That will produce a trimmed fastq file that contains the DNA reads. This is where `mutagenesis_visualization` kicks in. The following function `count_reads` will read your trimmed fastq file and count the number of times a DNA sequence is present. You will have to pass as inputs a `dna_sequence` and a `codon_list` with the codons that were used to make the point mutant library. If `savefile=True` , it will export the results to txt files. Below there is a prettified example of the output file.

```
# H-Ras dna sequence
hras_dnasequence: str =
↪'acggaatataagctggtggtggtgggcgccggcggtgtgggcaagagtgcgctgaccat'\
    +
↪'ccagctgatccagaaccattttgtggacgaatacgaccccactatagaggattcctaccggaagcaggtgg
↪'\
    + 'tcattgatggggagacgtgcctgttggacatcctg'

# Codons used to make the NNS library. I could also have used 'NNS'␣
↪and the package will use the NNS codons
codon_list: List[str] = [
    "GCC", "GCG", "TGC", "GAC", "GAG", "TTC", "GGC", "GGG", "CAC", "ATC
↪", "AAG",
    "CTC", "CTG", "TTG", "ATG", "AAC", "CCC", "CCG", "CAG", "CGC", "CGG
↪", "AGG",
    "TCC", "TCG", "AGC", "ACC", "ACG", "GTC", "GTG", "TGG", "TAC", "TAG
↪"
]

counts_wt: bool = False
start_position: int = 2

# Execute count reads
df_counts_pre, wt_counts_pre = count_reads(
    hras_dnasequence, HRAS_FASTQ, codon_list, counts_wt, start_
↪position)
```

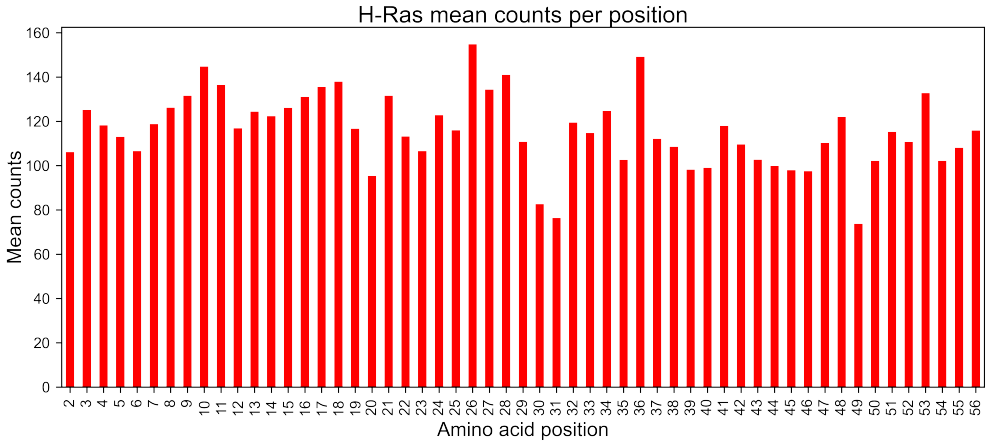
Codons	T	E	Y	K	L	V
GCC	1102	972	1172	725	577	554
GCG	990	788	647	578	401	575
TGC	837	612	710	547	508	1251
GAC	799	794	308	381	296	539
GAG	970	942	410	897	182	357
TTC	788	406	596	421	873	440
GGC	1127	702	800	502	291	663
GGG	2101	1216	1228	819	267	885

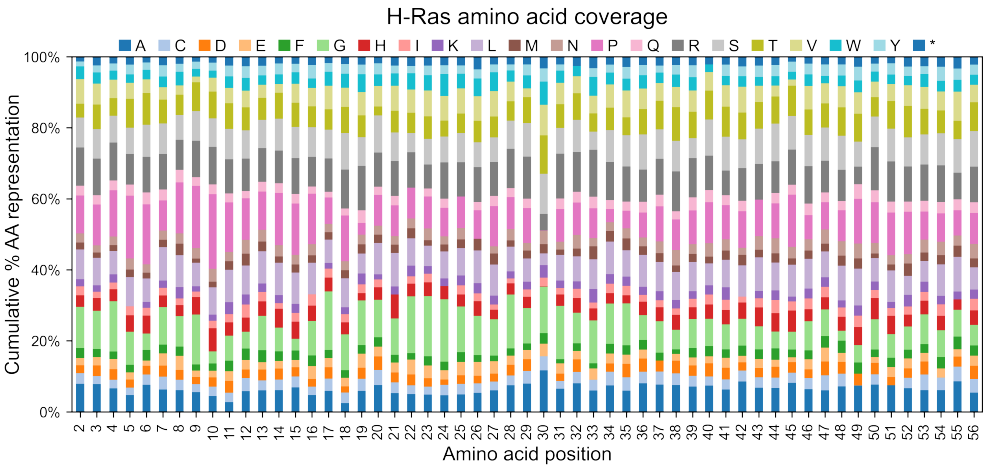
Create object of class `Counts`.

```
hras_obj = Counts(df_counts_pre, start_position = 2)
```

Once the reads have been counted, the method `mean_counts` can be used to evaluate the coverage by position. The method `library_representation` will tell you the percentage coverage of each amino acid per position.

```
hras_obj.mean_counts(title='H-Ras mean counts per position')

hras_obj.library_representation(title='H-Ras amino acid coverage')
```





Custom DNA list

Use a custom input DNA list. That way it does not matter if you are using NNS or you have second order mutations. Create a list of variants on your own, and the software will count the frequency of each of those variants on the fastq file you provide as an input. In the example non of the sequences we are specifying are found in the trimmed file, thus there are 0% of useful reads.

```
# Create your list of variants
variants: List[str] = [
    'acggaatataagctggtggtggtgggcgccggcggtgtgggcaagagtgcgctgaccat' +

 ↪'ccagctgatccagaaccattttgtggacgaatacgaccccactatagaggattcctaccggaagcaggtgg
 ↪' +
    'tcattgatggggagacgtgcctgttggacatcctg',
    'aaaaaatataagctggtggtggtgggcgccggcggtgtgggcaagagtgcgctgaccat' +

 ↪'ccagctgatccagaaccattttgtggacgaatacgaccccactatagaggattcctaccggaagcaggtgg
 ↪' +
    'tcattgatggggagacgtgcctgttggacatcctg',
    'ttttttataagctggtggtggtgggcgccggcggtgtgggcaagagtgcgctgaccat' +

 ↪'ccagctgatccagaaccattttgtggacgaatacgaccccactatagaggattcctaccggaagcaggtgg
 ↪' +
    'tcattgatggggagacgtgcctgttggacatcctg'
]

variants, totalreads, usefulreads = count_fastq(variants, HRAS_FASTQ)

# Evaluate how many variants in the fastq file were useful
print(
    '{}/{} useful reads ({}%)'.format(
```

(continues on next page)

(continued from previous page)

```
        str(usefulreads), str(totalreads),
        str(int(usefulreads / totalreads * 100))
    )
)
```

4.3.3 Calculate enrichment scores

Methods and functions reviewed in this section:

- *mutagenesis_visualization.Screen*
- *mutagenesis_visualization.calculate_enrichment()*

If you are performing a selection experiment, where you sequence your library before and after selection, you will need to calculate the enrichment score of each mutant. The function to do so is `calculate_enrichment`. This function allows for different parameters to tune how the data is muted and normalized.

In this example, we show two different ways of using `calculate_enrichment`. Note that the parameters of choice will have a say on the final result. In the example, the tonality of red of the two heatmaps is slightly different. A more detailed explanation of the parameters can be found in *Normalizing datasets*.

```
# Read counts from file (could be txt, csv, xlsx, etc...)
df_counts_pre: DataFrame = pd.read_excel(
    HRAS_GAPGEF_COUNTS,
    'R1_before',
    skiprows=1,
    index_col='Codons',
    usecols='E:FN',
    nrows=32
)

df_counts_sel: DataFrame = pd.read_excel(
    HRAS_GAPGEF_COUNTS,
    'R1_after',
    skiprows=1,
    index_col='Codons',
    usecols='E:FN',
    nrows=32
)
```

```
# Ras parameters to create an object

# Define protein sequence
```

(continues on next page)

(continued from previous page)

```
hras_sequence: str =
↪'MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEY'\
                +
↪'SAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQIKRVKDSDDVPMVLVGNKCDLAARTVES'\
                +
↪'RQAQDLARSYGIPYIETSAKTRQGVEDAFYTLVREIRQHKLRKLNPPDESGPG'

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position: int = 2

# Define secondary structure
secondary = [['L0'], ['β1'] * (9 - 1), ['L1'] * (15 - 9), ['α1'] * (25␣
↪- 15),
             ['L2'] * (36 - 25), ['β2'] * (46 - 36), ['L3'] * (48 -␣
↪46),
             ['β3'] * (58 - 48), ['L4'] * (64 - 58), ['α2'] * (74 -␣
↪64),
             ['L5'] * (76 - 74), ['β4'] * (83 - 76), ['L6'] * (86 -␣
↪83),
             ['α3'] * (103 - 86), ['L7'] * (110 - 103), ['β5'] * (116 -
↪ 110),
             ['L8'] * (126 - 116), ['α4'] * (137 - 126), ['L9'] * (140␣
↪- 137),
             ['β6'] * (143 - 140), ['L10'] * (151 - 143), ['α5'] *␣
↪(172 - 151),
             ['L11'] * (190 - 172)]

# Substitute Nan values with 0
fillna = 0
```

```
# Order of amino acids (from count_reads)
aminoacids_NNS: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# Different parameters can be used to calculate the enrichment scores.␣
↪They are described in the implementation section

# Zeroing using the median of the population, and not using stop␣
↪codons to correct.
frequencies = calculate_enrichment(
    aminoacids=aminoacids_NNS,
    pre_lib=df_counts_pre.iloc[:, :54],
```

(continues on next page)

(continued from previous page)

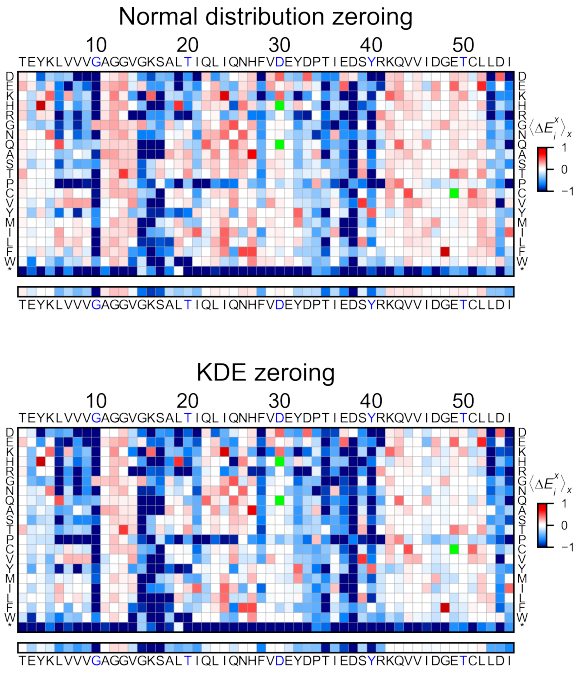
```
    post_lib=df_counts_sel.iloc[:, :54],
    zeroing_method='population',
    zeroing_metric ='median',
    norm_std=True,
    stopcodon=True,
    min_counts=25,
    min_countswt=100,
    mpop=2,
    mwt=2,
    infinite=3,
    std_scale=0.3
)

hras_example1 = Screen(
    np.array(frequencies), hras_sequence, aminoacids, start_position,␣
↪fillna,
    secondary
)

hras_example1.heatmap(title='Normal distribution zeroing', output_
↪file=None)

# Zeroing using the median of the population, and not using stop␣
↪codons to correct.
frequencies = calculate_enrichment(
    aminoacids=aminoacids_NNS,
    pre_lib=df_counts_pre.iloc[:, :54],
    post_lib=df_counts_sel.iloc[:, :54],
    zeroing_method='kernel',
    zeroing_metric ='median',
    norm_std=True,
    stopcodon=True,
    min_counts=25,
    min_countswt=100,
    mpop=2,
    mwt=2,
    infinite=3,
    std_scale=0.15
)

hras_example2 = Screen(
    np.array(frequencies), hras_sequence, aminoacids, start_position,␣
↪fillna,
    secondary
)
```

(continues on next page)

(continued from previous page)

```
hras_example2.heatmap(title='KDE zeroing', output_file=None)

# Note that the two heatmaps look quite similar but the red tonality␣
↪is slighly different. That is caused by
# small differences in zeroing the data.
```

Codons	T	E	Y	K	L	V
GCC	0.052	-0.233	-0.004	0.037	0.028	-0.198
GCG	-0.322	-0.193	-0.072	-0.090	0.234	-0.119
TGC	-0.072	-0.180	-0.098	0.124	-0.041	-0.191
GAC	0.161	-0.060	0.127	0.081	-0.256	-0.239
GAG	0.177	0.110	0.121	0.139		-0.180
TTC	-0.021	-0.165	-0.004	-0.116	-0.075	-0.067
GGC	-0.166	-0.151	-0.093	-0.025	-0.202	0.058
GGG	-0.059	-0.369	-0.076	0.159	0.082	-0.083



4.4 Normalizing datasets

This section will teach the different options to normalize the data using the function *mutagenesis_visualization.calculate_enrichment()* .

If you already have your own processing pipeline built, you can skip this section and go to the

(*Creating plots*) examples.

4.4.1 Import modules and load data

```
%matplotlib inline
from typing import List, Dict
import numpy as np
import pandas as pd
from pandas.core.frame import DataFrame

from mutagenesis_visualization import calculate_enrichment
from mutagenesis_visualization.main.utils.data_paths import HRAS_RBD_
↪COUNTS
from mutagenesis_visualization import Screen
```

Now let's add some information about Ras.

```
# Define protein sequence
hras_sequence: str =
↪'MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEY'\
                +
↪'SAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQIKRVKDSDDVPMVLVGNKCDLAARTVES'\
                +
↪'RQAQDLARSYGIPYIETSAKTRQGVEDAFYTLVREIRQHKLRKLNPPDESGPG'


# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position: int = 2

# Substitute Nan values with 0
fillna: int = 0
```

```
# List of sheets and columns to use
sheets_pre: List[str] = ['R1_before', 'R2_before', 'R3_before']
sheets_sel: List[str] = ['R1_after', 'R2_after', 'R3_after']
columns: List[str] = ['F:BG', 'BH:DK', 'DL:FN']
columns_wt: List[str] = ['A', 'B', 'C']

# Create dictionary with data. Loading 3 replicates, each of them is␣
↪divided into 3 pools
dict_pre, dict_sel, dict_pre_wt, dict_sel_wt = ({} for i in range(4))

# Read counts from file (could be txt, csv, xlsx, etc...)
for column, column_wt in zip(columns, columns_wt):
```

(continues on next page)

(continued from previous page)

```
    for sheet_pre, sheet_sel in zip(sheets_pre, sheets_sel):
        # Pre counts
        label_pre = str(sheet_pre + '_' + column_wt)
        dict_pre[label_pre] = pd.read_excel(
            HRAS_RBD_COUNTS, sheet_pre, skiprows=1, usecols=column,␣
↪nrows=32
        )
        # Pre counts wild-type alleles
        dict_pre_wt[label_pre] = pd.read_excel(
            HRAS_RBD_COUNTS, sheet_pre, usecols=column_wt
        )

        # Sel counts
        label_sel = str(sheet_sel + '_' + column_wt)
        dict_sel[label_sel] = pd.read_excel(
            HRAS_RBD_COUNTS, sheet_sel, skiprows=1, usecols=column,␣
↪nrows=32
        )
        # Sel counts wild-type alleles
        dict_sel_wt[label_sel] = pd.read_excel(
            HRAS_RBD_COUNTS, sheet_sel, usecols=column_wt
        )
```

4.4.2 Calculate log10 enrichment

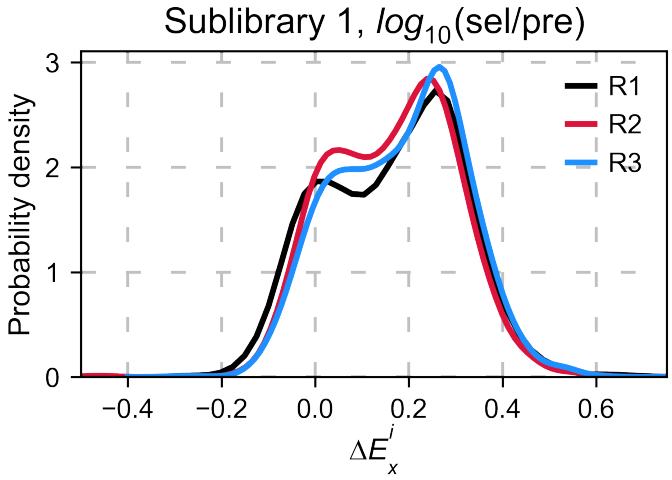
Now we are going to calculate the log10(sel/pre) for the sublibrary 1 of each replicate and plot a histogram. The resulting distribution is bimodal, and because the three replicates have a similar number of counts ratios, their center is overlapping. However, because we have not normalized by the number of counts, and there are more counts in the selected than in the pre-selected population, the center is >0.

```
# Auxiliar function to convert +-inf values to an arbitrary number (ie␣
↪+-2)
def _replace_inf(df: DataFrame) -> DataFrame:
    df.replace(to_replace=np.inf, value=2, inplace=True)
    df.replace(to_replace=-np.inf, value=-2, inplace=True)
    return df


aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')
enrichment = {}

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
```

(continues on next page)

(continued from previous page)

```
                                list(dict_sel.keys())[:3]):
    # log 10
    enrichment_log10 = (np.log10(dict_sel[sel_key] / dict_pre[pre_
↪key]))
    enrichment_log10['aminoacids'] = aminoacids
    enrichment_log10.set_index(['aminoacids'], inplace=True)
    enrichment[pre_key[:2]] = _replace_inf(enrichment_log10)

# Create objects
hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)

hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"], title=r'$log_{10}$' + '(sel/pre)', xscale=(-1, 1))
```



4.4.3 Centering the data (zeroing)

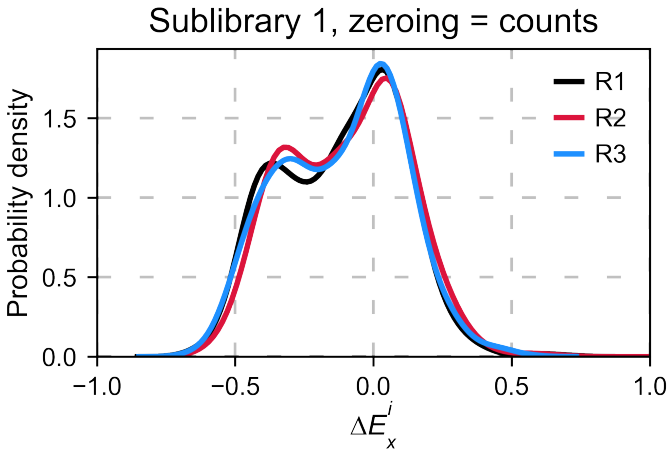
Functions used in this section:

- *mutagenesis_visualization.main.kernel.kernel.Kernel*
- *mutagenesis_visualization.calculate_enrichment()*

Counts normalization

Normalizing by the number of counts improves normalization. Now the population center is closer to 0. To do so, set `zeroing_method='counts'`.

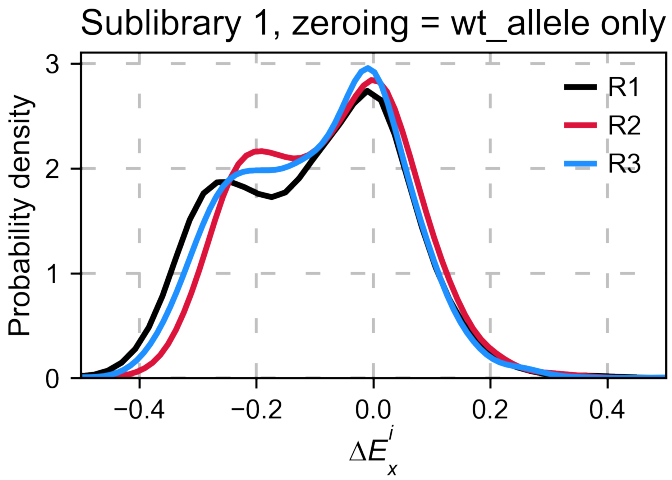
```
enrichment = {}
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')
# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # Enrichment
    enrichment[pre_key[:2]] = calculate_enrichment(
        aminoacids, dict_pre[pre_key], dict_sel[sel_key], zeroing_
↪method='counts', stopcodon=False
    )

# Plot histogram and KDE
aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)

hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"], title='zeroing_method = counts', xscale=(-1, 0.5))
```



Wt allele

Another way we can normalize is by using an internal reference such as a particular mutant. In the following example we will use the wild-type allele. If the assay that you are using is noisy, relying on a single data point for normalizing will result in high variance. The package does not include this option because it may lead to errors. Here we are showing how it would be done by hand. In this example, it works fine. But in other datasets we have, it has been a source of error.

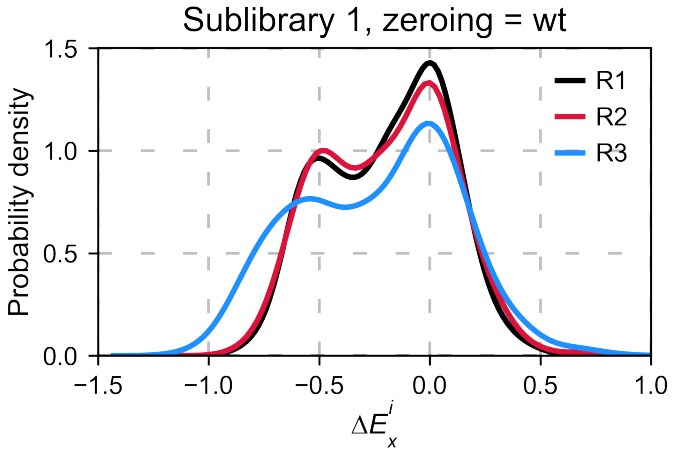
```
# calculate log10 enrichment for each replicate

aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')
enrichment = {}

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # log 10
    wt_ratio = np.log10(
        dict_sel_wt[sel_key]['wt 2-56'][1] / dict_pre_wt[pre_key]['wt␣
↪2-56'][1]
    )
    enrichment_log10 = np.log10(
        dict_sel[sel_key] / dict_pre[pre_key]
    ) - wt_ratio
    enrichment_log10['aminoacids'] = aminoacids
    enrichment_log10.set_index(['aminoacids'], inplace=True)
    enrichment[pre_key[:2]] = _replace_inf(enrichment_log10)

hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)
hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"], title='zeroing_method = wt allele only', xscale=(-0.5,␣
↪0.5))
```



Distribution of synonymous wt alleles

In our experience, it is better to use the median/mode/mean of the synonymous wild-type population because there is less variance. `calculate_enrichment` has such an options by using `zeroing_method='wt'` and then `zeroing_metric ='median', 'mean' or 'mode'`.

```
enrichment = {}
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # Enrichment
    enrichment[pre_key[:2]] = calculate_enrichment(
        aminoacids,
        dict_pre[pre_key],
        dict_sel[sel_key],
        dict_pre_wt[pre_key],
        dict_sel_wt[sel_key],
        zeroing_method='wt',
        zeroing_metric ='mode',
        stopcodon=False
    )

aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
 ↪position, fillna,
)
hras_object.kernel(show_replicates=True, title='Sublibrary 1, zeroing_
 ↪method = wt', xscale=(-1.5, 1))
```

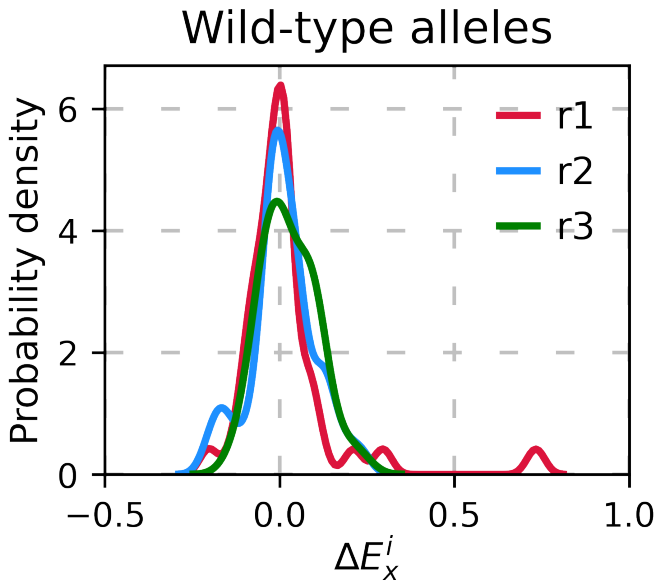


Wt alleles observation

If the population of synonymous wild-type alleles (alleles that are wild-type at a protein level, but not at a DNA level) is small, the distribution of this variants may have high variance from sample to sample. Also, you will notice that not all wild-type alleles are neutral. The spread of these alleles gives a sense of the noise in the experiment.

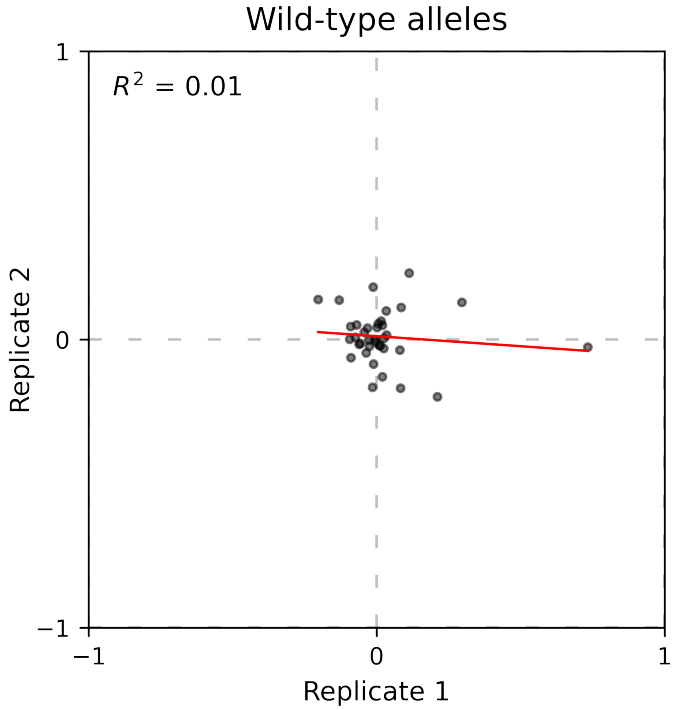
At least for the following data, there is no correlation between the performance of wild-type alleles in different replicates, suggesting that the higher or lower enrichment scores are caused by noise and not a fitness difference caused by changes in protein expression.

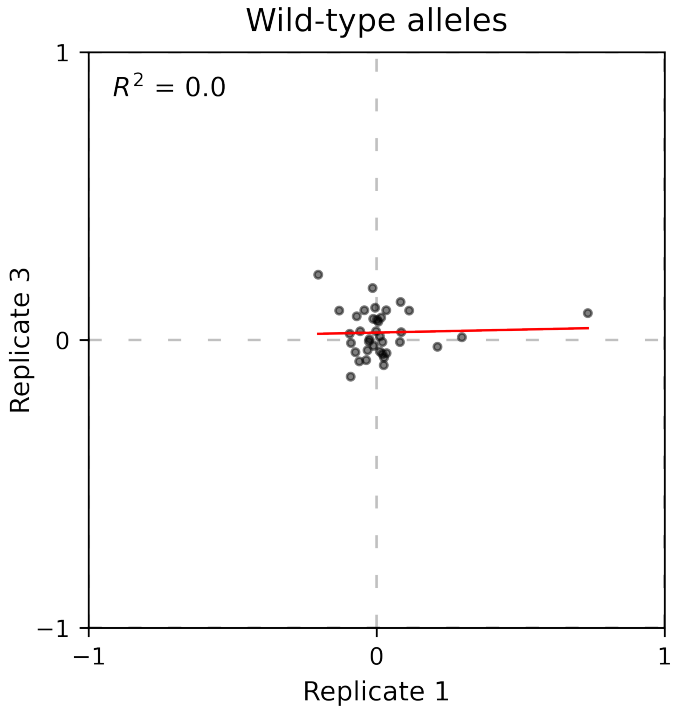
```
hras_object.kernel(show_replicates=True, wt_counts_only=True,title=
↪'Wild-type alleles', kernel_colors=['k', 'crimson', 'dodgerblue', 'g
↪', 'silver'], xscale=(-0.5, 1), output_file="docs/images/exported_
↪images/hras_wildtype_distribution.png")
```

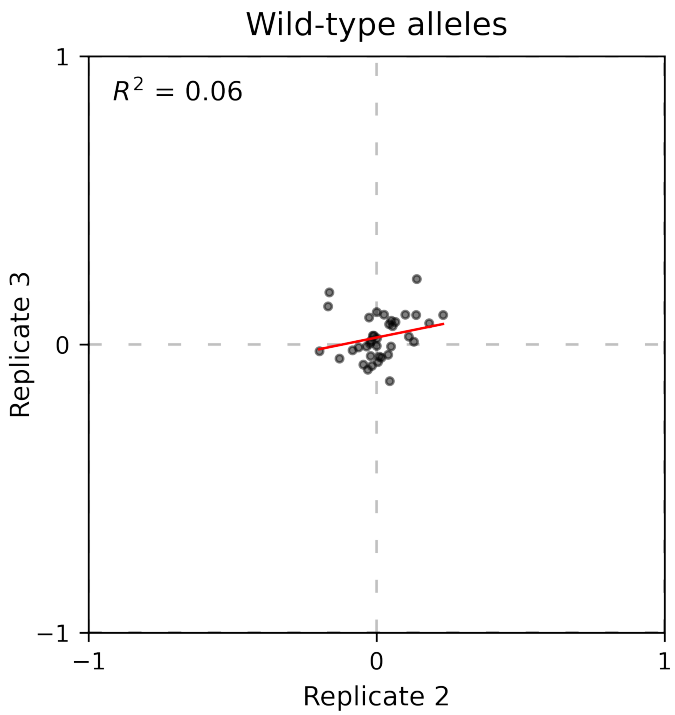


Perform the scatter plots:

```
hras_object.scatter_replicates(wt_counts_only=True,title='Wild-type␣
↪alleles', xscale=(-1, 1), yscale=(-1, 1), output_file="docs/images/
↪exported_images/hras_wildtype_scatter.png")
```


Wild-type alleles
R^2 = 0.01
Replicate 2
Replicate 1
1
0
−1
−1
0
1


Wild-type alleles
R^2 = 0.0
Replicate 3
Replicate 1
1
0
−1
−1
0
1



Distribution of mutants

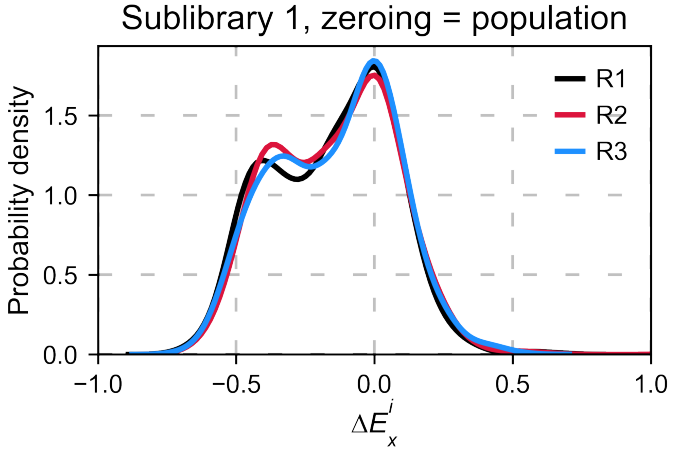
An alternative option to normalize the data is to use the mean/median/mode of the population to some specific number such as zero. To do so, use `zeroing_method='population'`. The parameters of the distribution will be calculated assuming a gaussian distribution. Not only the three replicates are centered, but also they have the same spread.

```
enrichment = {}
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # Enrichment
    enrichment[pre_key[:2]]  = calculate_enrichment(
        aminoacids,
        dict_pre[pre_key],
        dict_sel[sel_key],
        zeroing_method='population',
        zeroing_metric ='mode',
        stopcodon=False
    )

aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')
```

(continues on next page)

(continued from previous page)

```
hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)
hras_object.kernel(show_replicates=True, title='zeroing_method =␣
↪population', xscale=(-1, 1))
```



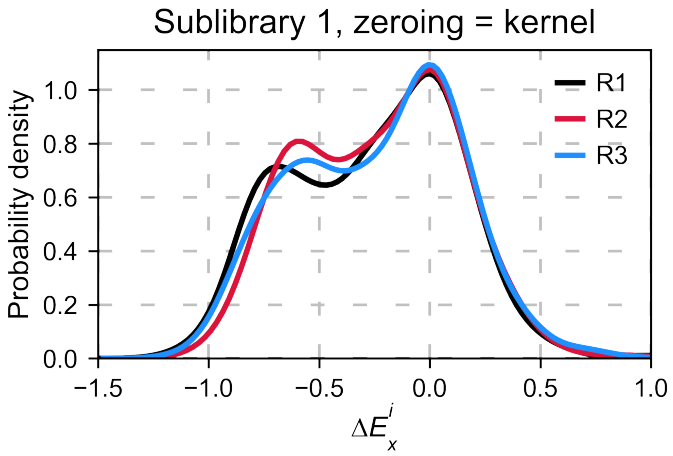
A variant of the previous method is to calculate the kernel density estimate using `zeroing_method='kernel'`. This option centers the population using the mode of the KDE. If the data is bimodal, it will select the main peak. Furthermore, it will use the standard deviation of the main peak to scale the data. This method is useful when you have split your library into multiple pools because it will not only center the data properly but also do scale the data so each pool main peak has the same standard deviation. Results are quite similar to setting `zeroing_method='population'` and `zeroing_metric ='mode'`.

```
enrichment = {}
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # Enrichment
    enrichment[pre_key[:2]] = calculate_enrichment(
        aminoacids, dict_pre[pre_key], dict_sel[sel_key], zeroing_
↪method='kernel', stopcodon=False
    )

aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

hras_object: Screen = Screen(
```

(continues on next page)

(continued from previous page)

```
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)
hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"],  title='zeroing method = kernel', xscale=(-1.5,1))
```



4.4.4 Baseline subtraction

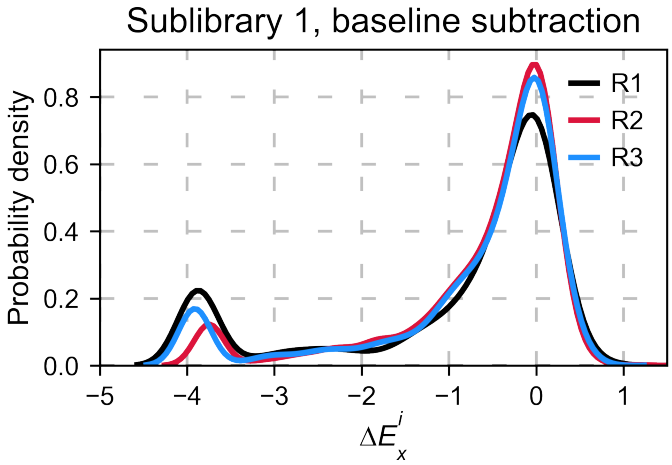
Including stop codons in the library can be of great use because it gives a control for basal signal in your assay. The algorithm has the option to apply a baseline subtraction. The way it works is it sets the stop codons counts of the selected population to 0 (baseline) and subtracts the the baseline signal to every other mutant. To use this option, set `stopcodon=True`. You will notice that it get rids of the shoulder peak, and now the distribution looks unimodal with a big left shoulder.

```
enrichment = {}
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# calculate log10 enrichment for each replicate
for pre_key, sel_key in zip(list(dict_pre.keys())[:3],
                            list(dict_sel.keys())[:3]):
    # Enrichment
    enrichment[pre_key[:2]] = calculate_enrichment(
        aminoacids, dict_pre[pre_key], dict_sel[sel_key], zeroing_
↪method='kernel', stopcodon=True
    )

aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

hras_object: Screen = Screen(
    list(enrichment.values()), hras_sequence, aminoacids, start_
↪position, fillna,
```

(continues on next page)

(continued from previous page)

```
)
hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"], title='stop codon correction', xscale=(-5, 1.5))
```



4.4.5 Scaling

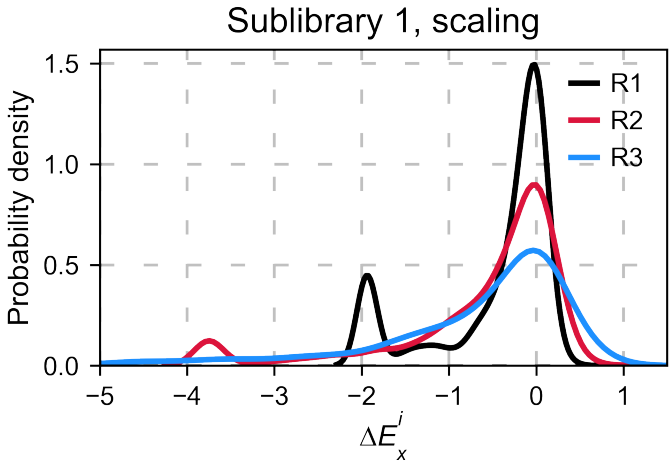
By now you probably have realized that different options of normalization affect to the spread of the data. The rank between each mutant is unchanged between the different methods, so it is a matter of multiplying/dividing by a scalar to adjust the data spread. Changing the value of the parameter `std_scale` will do the job. You will probably do some trial an error until you find the right value. In the following example we are changing the `std_scale` parameter for each of the three replicates shown. Note that the higher the scalar, the higher the spread.

```
enrichment_scalar = {}
scalars: List[str] = [0.1, 0.2, 0.3]
aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*')

# calculate log10 enrichment for each replicate
for pre_key, sel_key, scalar in zip(list(dict_pre.keys())[:3],
                                    list(dict_sel.keys())[:3],␣
↪scalars):
    # Enrichment
    enrichment_log10 = calculate_enrichment(
        aminoacids,
        dict_pre[pre_key],
        dict_sel[sel_key],
        zeroing_method='kernel',
        stopcodon=True,
        std_scale=scalar
```

(continues on next page)

(continued from previous page)

```
    )
    enrichment_scalar[pre_key[:2]] = enrichment_log10


aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

hras_object: Screen = Screen(
    list(enrichment_scalar.values()), hras_sequence, aminoacids, start_
↪position, fillna,
)
hras_object.kernel(show_replicates=True, kernel_color_replicates = ["b
↪", "r", "g"], title='scaling', xscale=(-5, 1.5))
```



4.4.6 Multiple sublibraries

In our own research projects, where we have multiple DNA pools, we have determined that the combination of parameters that best suit us it to the wild-type synonymous sequences to do a first data normalization step. Then use `zeroing_method = 'kernel'` to zero the data and use `stopcodon=True` in order to determine the baseline level of signal. You may need to use different parameters for your purposes. Feel free to get in touch if you have questions regarding data normalization.

```
# Labels
labels: List[str] = ['Sublibrary 1', 'Sublibrary 2', 'Sublibrary 3']
zeroing_options: List[str] = ['population', 'counts', 'wt', 'kernel']
title: str = 'Rep-A sublibraries, zeroing_method = '

# xscale
xscales = [(-2, 1), (-2.5, 0.5), (-3.5, 1.5), (-3.5, 1.5)]
```

(continues on next page)

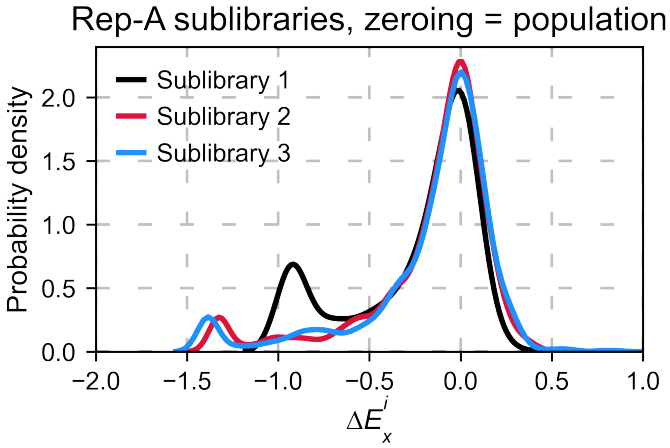
(continued from previous page)

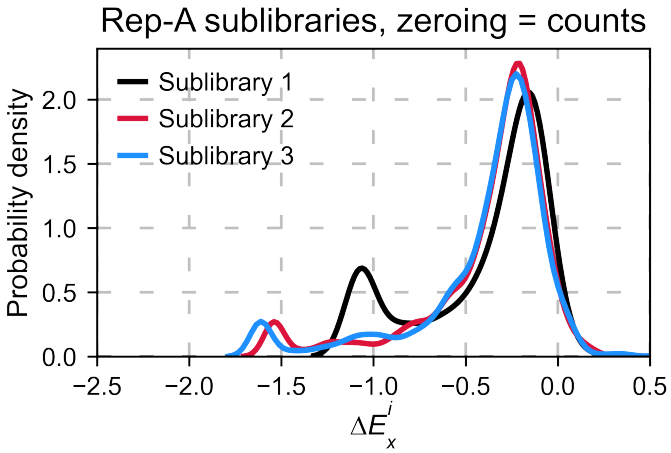
```
# declare dictionary
enrichment_lib = {}
df_lib = {}

for option, xscale in zip(zeroing_options, xscales):
    for pre_key, sel_key, label in zip(list(dict_pre.keys())[::3],
                                       list(dict_sel.keys())[::3],␣
↪labels):
        aminoacids: List[str] = list('AACDEFGGHIKLLLMNPPQRRRSSSTTVVWY*
↪')

        # log 10
        enrichment_lib[label]  = DataFrame(calculate_enrichment(
            aminoacids,
            dict_pre[pre_key],
            dict_sel[sel_key],
            dict_pre_wt[pre_key],
            dict_sel_wt[sel_key],
            zeroing_method=option,
            zeroing_metric ='mode',
            stopcodon=True,
            infinite=2
        ))

    # Concatenate sublibraries and store in dict
    df_lib[option] = pd.concat([
        enrichment_lib['Sublibrary 1'], enrichment_lib['Sublibrary 2'],
        enrichment_lib['Sublibrary 3']
    ],ignore_index=True, axis=1)


    # Plot
    aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

    hras_sublibrary1: Screen = Screen(
        enrichment_lib['Sublibrary 1'], hras_sequence, aminoacids,␣
↪start_position, fillna,
    )
    hras_sublibrary2: Screen = Screen(
        enrichment_lib['Sublibrary 2'], hras_sequence, aminoacids,␣
↪start_position, fillna,
    )
    hras_sublibrary3: Screen = Screen(
        enrichment_lib['Sublibrary 3'], hras_sequence, aminoacids,␣
↪start_position, fillna,
```

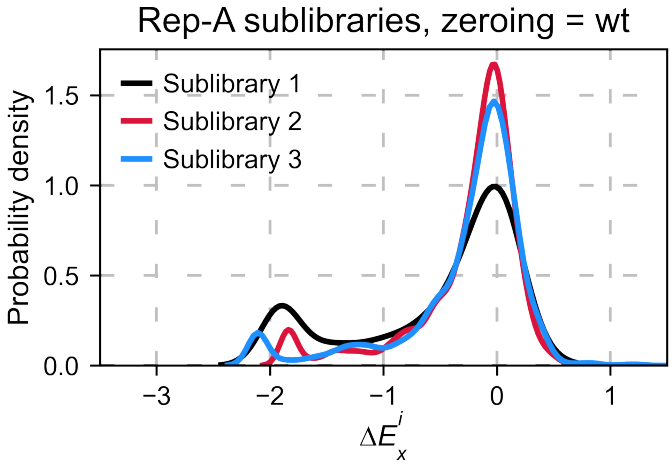
(continues on next page)

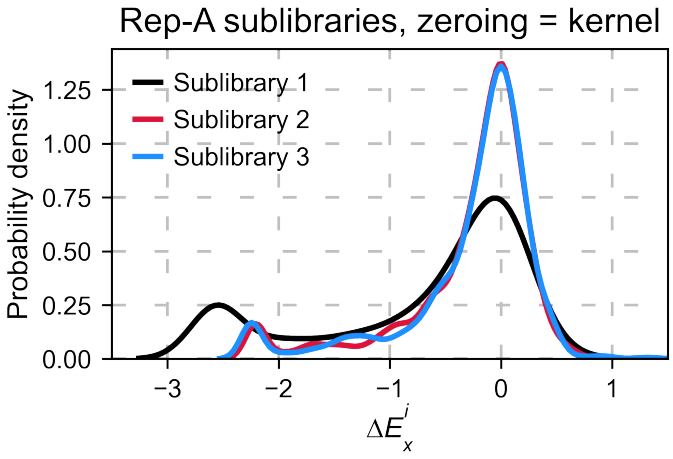
(continued from previous page)

```
    )
    hras_sublibrary1.multiple_kernel([hras_sublibrary2, hras_
↪sublibrary3], label_kernels = labels, title=title + option,␣
↪xscale=xscale)
```









4.4.7 Heatmaps

Function and class used in this section:

- *mutagenesis_visualization.Screen*
- *mutagenesis_visualization.main.heatmaps.heatmap.Heatmap()*

We are going to evaluate how does the heatmap of produced by each of the normalization methods. We are not going to scale the data, so some heatmaps may look more washed out than others. That is not an issue since can easily be changed by using `std_scale`.

```
# First we need to create the objects

# Define protein sequence
hras_sequence: str =
↪'MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEY'\
                +
↪'SAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQIKRVKDSDDVPMVLVGNKCDLAARTVES'\
                +
↪'RQAQDLARSYGIPYIETSAKTRQGVEDAFYTLVREIRQHKLRKLNPPDESGPG'

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position: int = 2

# Create objects
objects: Dict[str, Screen] = {}
for key, value in df_lib.items():
```

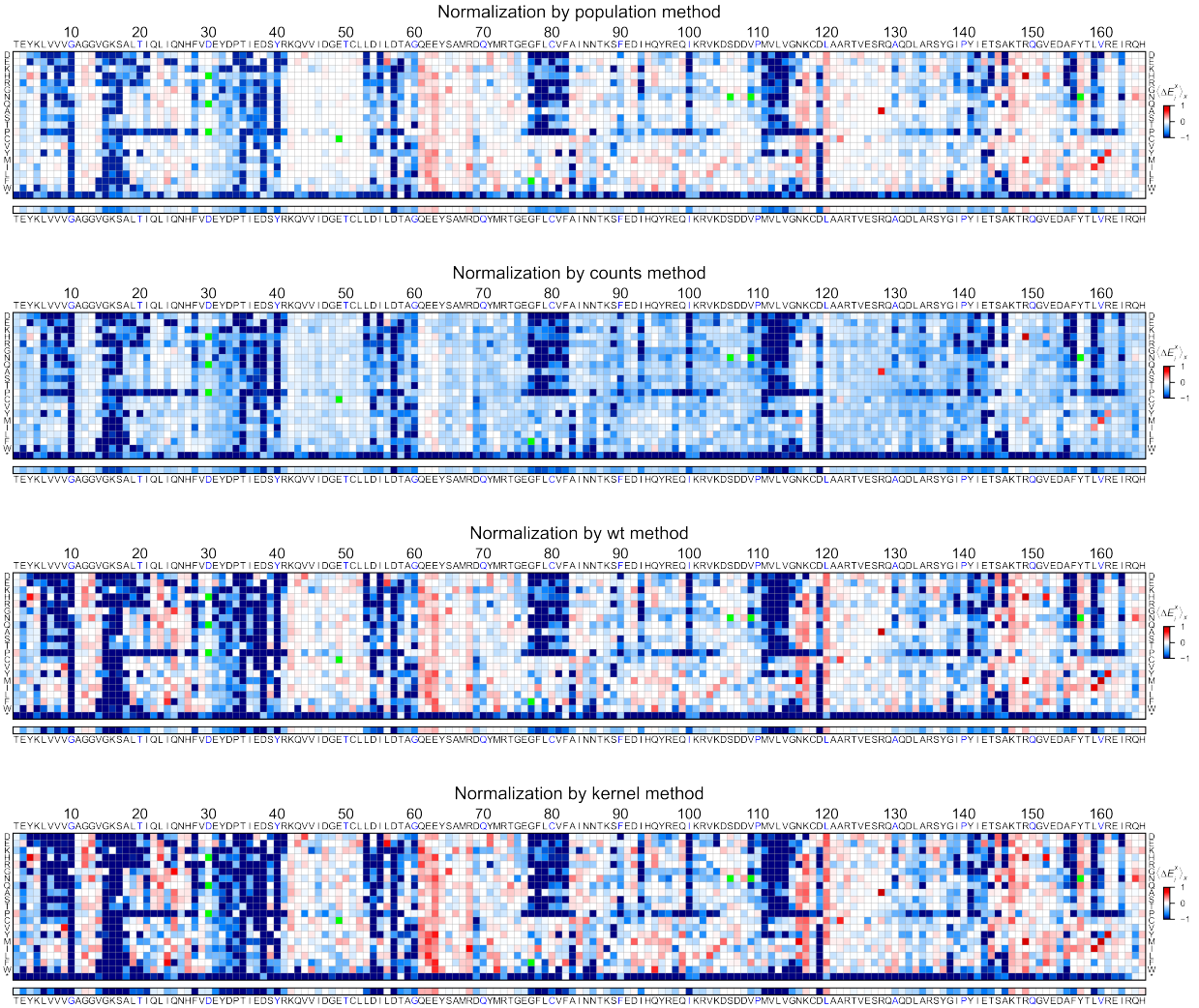
(continues on next page)

(continued from previous page)

```
    temp = Screen(value, hras_sequence, aminoacids, start_position)
    objects[key] = temp
```

Now that the objects are created and stored in a dictionary, we will use the method `object.heatmap`. You will note that the first heatmap ("population") looks a bit washed out. If you look at the kernel distribution, the spread is smaller. The "kernel" and "wt" heatmaps look almost identical, while the "counts" heatmap looks all blue. This is caused by the algorithm not being able to center the data properly, and everything seems to be loss of function. That is why it is important to select the method of normalization that works with your data.

```
titles: List[str] = ['population', 'counts', 'wt', 'kernel']

# Create objects
for obj, title in zip(objects.values(), titles):
    obj.heatmap(title='Normalization by ' + title + ' method')
```



4.5 Creating heatmaps

This section shows how to use the mutagenesis_visualization package. The plotting functions can be used regardless of how you process your data. For the examples, we are using two datasets that are derived from Pradeep's legacy.[1]

4.5.1 Import modules

```
# running locally, if you pip install then you just have to import the␣
↪module
%matplotlib inline
from typing import List
import numpy as np
import matplotlib as plt
import copy
from mutagenesis_visualization import Screen
from mutagenesis_visualization.main.utils.data_paths import HRAS_RBD_
↪COUNTS_CSV, HRAS_GAPGEF_COUNTS_CSV
```

4.5.2 Create object of class Screen

Class reviewed in this section:

- `mutagenesis_visualization.main.classes.screen.Screen`

In order to create plots, the first step is to create a `Screen.object`. The enrichment scores will be passed using the parameter `dataset` . The protein sequence `sequence` and the amino acid substitutions order `aminoacids` need to be defined for the object to be created. Adding the secondary structure `secondary` is optional, but without it some plots will not work. In this example, we are importing two datasets and creating two objects named `hras_GAPGEF` and `hras_RBD`.

```
# Load enrichment scores. This is how you would load them from a local␣
↪file.
hras_enrichment_GAPGEF = np.genfromtxt(HRAS_GAPGEF_COUNTS_CSV,␣
↪delimiter=',')
hras_enrichment_RBD = np.genfromtxt(HRAS_RBD_COUNTS_CSV, delimiter=',')

# Define protein sequence
hras_sequence: str =
↪'MTEYKLVVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEY'\
```

(continues on next page)

[1] Bandaru, P., Shah, N. H., Bhattacharyya, M., Barton, J. P., Kondo, Y., Cofsky, J. C., ... Kuriyan, J. (2017). Deconstruction of the Ras switching cycle through saturation mutagenesis. ELife, 6. DOI: 10.7554/eLife.27810

(continued from previous page)

```
                +
↪'SAMRDQYMRTGEGFLCVFAINNTKSFEDIHQYREQIKRVKDSDDVPMVLVGNKCDLAARTVES'\
                +
↪'RQAQDLARSYGIPYIETSAKTRQGVEDAFYTLVREIRQHKLRKLNPPDESGPG'

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids: List[str] = list('ACDEFGHIKLMNPQRSTVWY*')

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position: int = 2

# Define secondary structure
secondary = [['L0'], ['β1'] * (9 - 1), ['L1'] * (15 - 9), ['α1'] * (25␣
↪- 15),
             ['L2'] * (36 - 25), ['β2'] * (46 - 36), ['L3'] * (48 -␣
↪46),
             ['β3'] * (58 - 48), ['L4'] * (64 - 58), ['α2'] * (74 -␣
↪64),
             ['L5'] * (76 - 74), ['β4'] * (83 - 76), ['L6'] * (86 -␣
↪83),
             ['α3'] * (103 - 86), ['L7'] * (110 - 103), ['β5'] * (116 -
↪ 110),
             ['L8'] * (126 - 116), ['α4'] * (137 - 126), ['L9'] * (140␣
↪- 137),
             ['β6'] * (143 - 140), ['L10'] * (151 - 143), ['α5'] *␣
↪(172 - 151),
             ['L11'] * (190 - 172)]

# Substitute Nan values with 0
fillna: int = 0

# Create objects
hras_GAPGEF: Screen = Screen(
    hras_enrichment_GAPGEF, hras_sequence, aminoacids, start_position,␣
↪fillna,
    secondary
)
hras_RBD: Screen = Screen(
    hras_enrichment_RBD, hras_sequence, aminoacids, start_position,␣
↪fillna,
    secondary
)
```

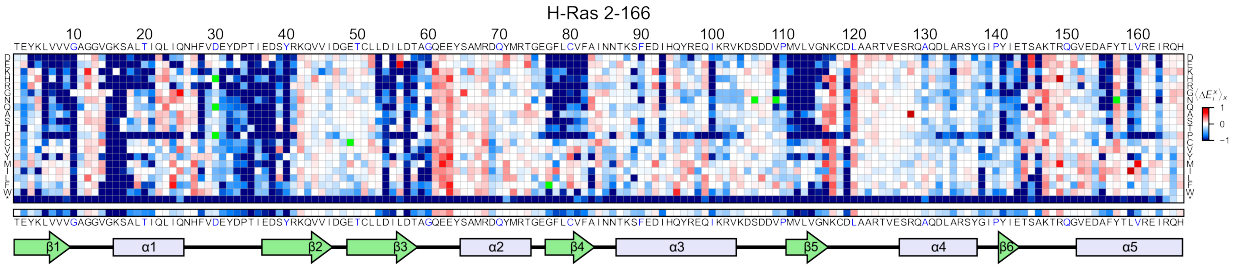
4.5.3 Heatmaps

Methods reviewed in this section:

- *mutagenesis_visualization.main.heatmaps.heatmap.Heatmap()*
- *mutagenesis_visualization.main.heatmaps.heatmap_rows.HeatmapRows()*
- mutagenesis_visualization.main.heatmaps.heatmap.columns.HeatmapColumns()
- *mutagenesis_visualization.main.heatmaps.miniheatmap.Miniheatmap()*

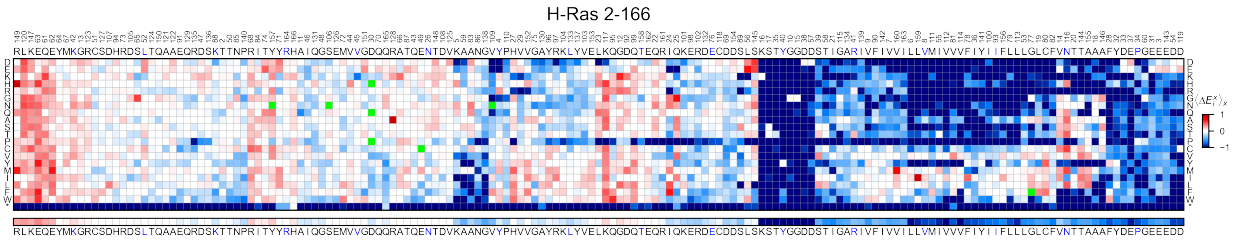
Once the object `hras_RBD` is created, we will plot a heatmap of the enrichment scores using the method `object.heatmap`.

```
# Create full heatmap
hras_RBD.heatmap(title='H-Ras 2-166', show_cartoon=True)
```



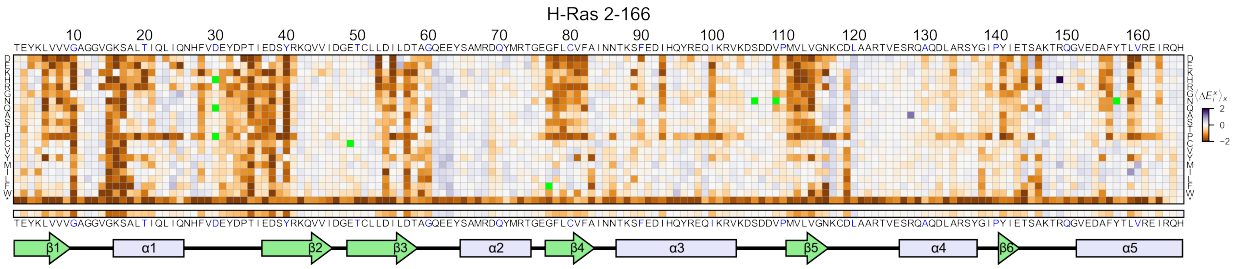
If you set the parameter `hierarchical=True`, it will sort the columns using hierarchical clustering

```
hras_RBD.heatmap(title='H-Ras 2-166', hierarchical=True, output_
↪file=None)
```



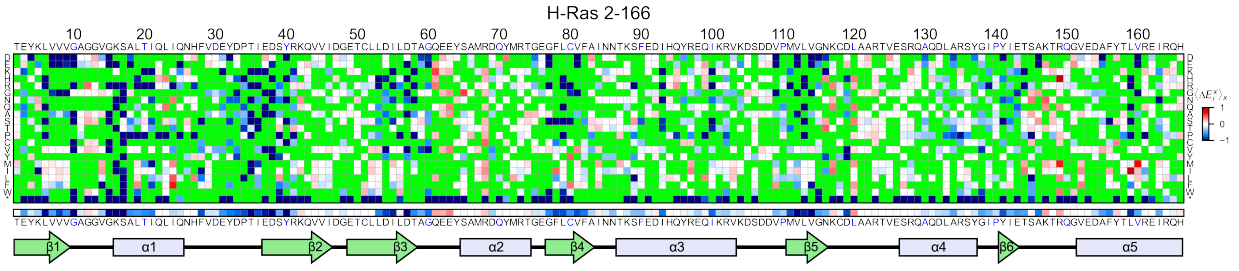
You can change the scale and the color map using the parameters `colorbar_scale` and `colormap`. You can also mask self-substitutions (ie T2T) by setting `mask_selfsubstitutions=True`. The noise in the assay may cause self-substitutions to have a score different than 0, which may confuse the reader. If you use this masking, please make sure that there is no systematic error related to the centering of the data.

```
# Load a color map from matplotlib
colormap = copy.copy((plt.cm.get_cmap('PuOr')))

# Change scale and colormap
hras_RBD.heatmap(
    mask_selfsubstitutions=True,
    title='H-Ras 2-166',
    colorbar_scale=(-2, 2),
    colormap=colormap,
    show_cartoon=True,
)
```



If you set the parameter `show_snv=True`, the algorithm will color green every mutation that is not a single nucleotide variant (SNV) of the wild-type protein. You will notice how many mutations are not accessible through a nucleotide change. This option may be useful to you so you can quickly evaluate which mutations are accessible through random DNA mutations. In the example of Ras, the frequency of non-SNV substitutions at residues 12 and 13 is dramatically lower.

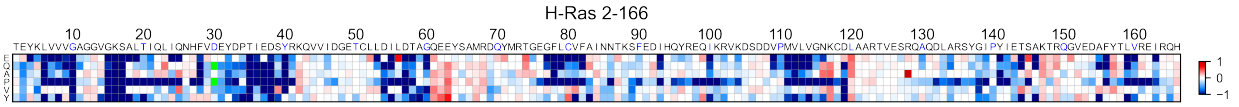
```
# Create full heatmap showing only SNV mutants
hras_RBD.heatmap(
    title='H-Ras 2-166', show_cartoon=True, show_snv=True)
```



We can slice the full heatmap by either showing only some columns or some rows. To show only a few amino acid mutational profiles (rows), we will use the method `object.heatmap_rows`. Note that we need to specify which amino acids to show with `selection`.

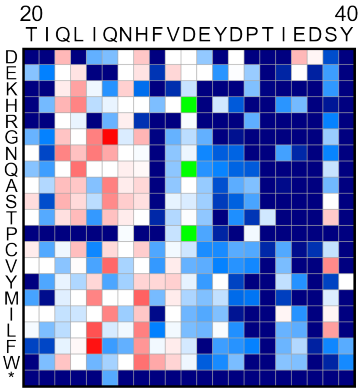
4.5.4 Heatmap slices

```
# Create heatmap of selected aminoacid substitutions
hras_RBD.heatmap_rows(
    title='H-Ras 2-166',
    selection=['E', 'Q', 'A', 'P', 'V', 'Y'],
)
```



If we want to display only a few positions in the protein (columns), we will use the method `object.heatmap_columns`. The parameter `segment` will indicate which are the contigous columns to show.

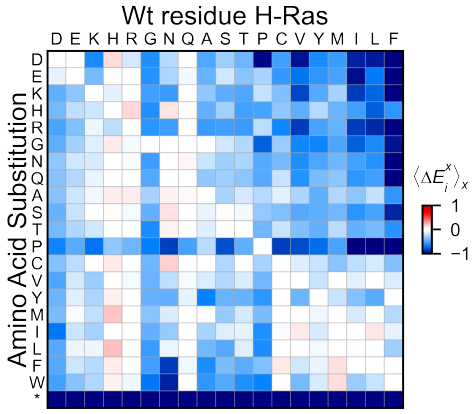
```
# Create a heatmap of a subset region in the protein
hras_RBD.heatmap_columns(segment=[20, 40])
```



4.5.5 Miniheamap

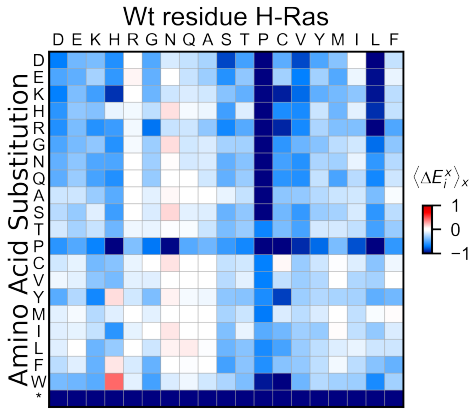
A summarized heatmap can also be generated. It is useful to evaluate global trends in the data. The command to use is `object.miniheatmap`.

```
# Condensed heatmap
hras_RBD.miniheatmap(title='Wt residue H-Ras')
```



Now lets look at the effect of having a certain residue in front the mutated residue. For instance, the column of prolines is the average of all the columns that had a proline in the n-1 position. To accomplish this, `set offset=-1`.

```
# Condensed heatmap offset no background correction
hras_RBD.miniheatmap(
    title='Wt residue H-Ras',
    offset=-1,
    background_correction=False,
)
```



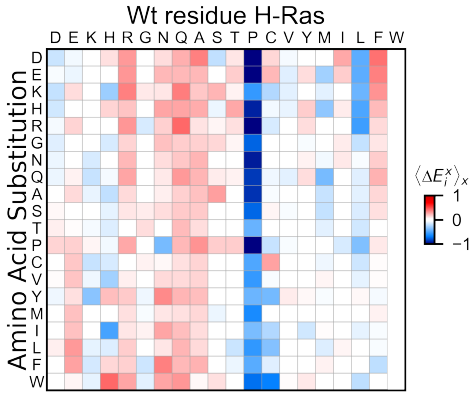
Now lets do a background correction by setting `background_correction=True`. To the calculated values, it will subtract the mean enrichment score for every substitution type. In the example, proline is the only residues than wen situated before the mutation, it seems to have a detrimental effect.

```
# Condensed heatmap offset with background correction
hras_RBD.miniheatmap(
    title='Wt residue H-Ras',
    offset=-1,
    background_correction=True,
```

(continues on next page)

(continued from previous page)

```
)
```



4.5.6 Reference

4.6 Creating plots

This section continues showing how to do different types of plots.

4.6.1 Import modules

```
# running locally, if you pip install then you just have to import the␣
↪module
%matplotlib inline
from pandas.core.frame import DataFrame
import numpy as np
import pandas as pd
from mutagenesis_visualization.main.demo.demo_objects import␣
↪DemoObjects
DEMO_OBJECTS:DemoObjects = DemoObjects()
hras_rbd = DEMO_OBJECTS.hras_rbd
hras_gapgef = DEMO_OBJECTS.hras_gapgef
```

4.6.2 Histogram, scatter and more

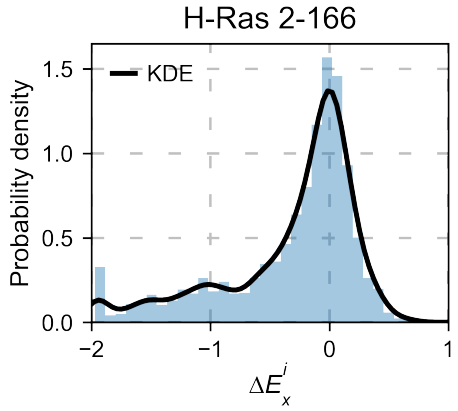
Classes reviewed in this section:

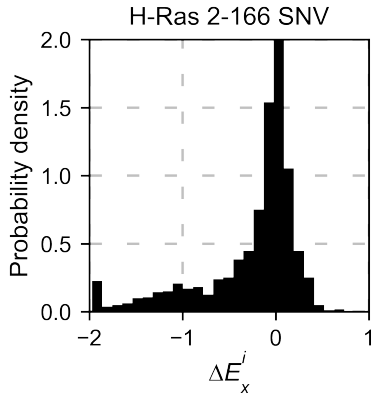
- *mutagenesis_visualization.main.kernel.kernel.Kernel*
- *mutagenesis_visualization.main.kernel.histogram.Histogram*

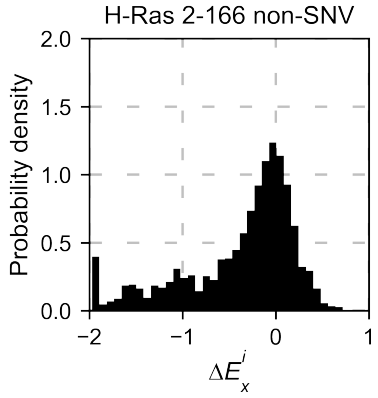
- *mutagenesis_visualization.main.kernel.sequence_differences.SequenceDifferences*
- *mutagenesis_visualization.main.scatter.scatter.Scatter*
- *mutagenesis_visualization.main.other_stats.rank.Rank*
- *mutagenesis_visualization.main.other_stats.cumulative.Cumulative*

There are different tools to analyze the data. The package can plot the kernel density estimation (`object.kernel`). There is the option to fit other functions to the data (see Implementation for more). You could also only plot a histogram (`object.histogram`). For the histograms, we can select to plot only the single nucleotide variants (SNVs) or the non-SNVs. In the example, it actually changes the shape of the population. Non-SNVs are more sensitive to mutations than SNVs because there is a higher proportion of non-conservative amino acid replacements.

```
# Plot kernel dist using sns.distplot.
hras_rbd.kernel(
    title='H-Ras 2-166', xscale=[-2, 1]
)

# Plot histogram of SNVs
hras_rbd.histogram(
    population='SNV', title='H-Ras 2-166 SNV', xscale=[-2, 1]
)

# Plot histogram of non-SNVs
hras_rbd.histogram(
    population='nonSNV',
    title='H-Ras 2-166 non-SNV',
    xscale=[-2, 1],
)
```







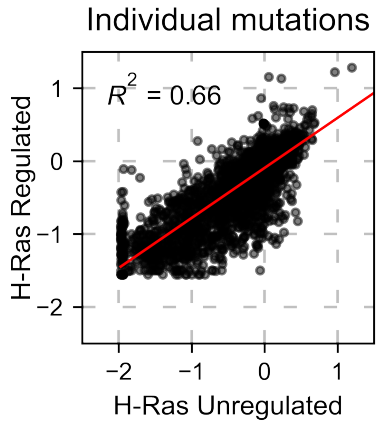
If you have multiple datasets and want to compare them, you can do it with the method `object.scatter`. We give the option to do the comparison at a mutation by mutation level `mode=pointmutant`, or at a position level `mode=mean`.

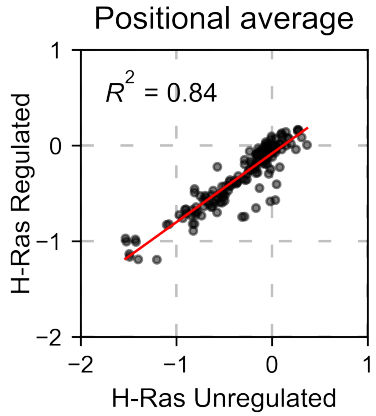
```
# Plot a scatter plot of each mutation
hras_rbd.scatter(
    hras_gapgef,
    title='Individual mutations',
    mode='pointmutant',
    xscale=(-2.5, 1.5),
    yscale=(-2.5, 1.5),
    x_label='H-Ras Unregulated',
    y_label='H-Ras Regulated',
)

# Plot a scatter plot of the mean position
hras_rbd.scatter(
    hras_gapgef,
    title='Positional average',
    mode='mean',
    xscale=(-2, 1),
    yscale=(-2, 1),
```

(continues on next page)

(continued from previous page)

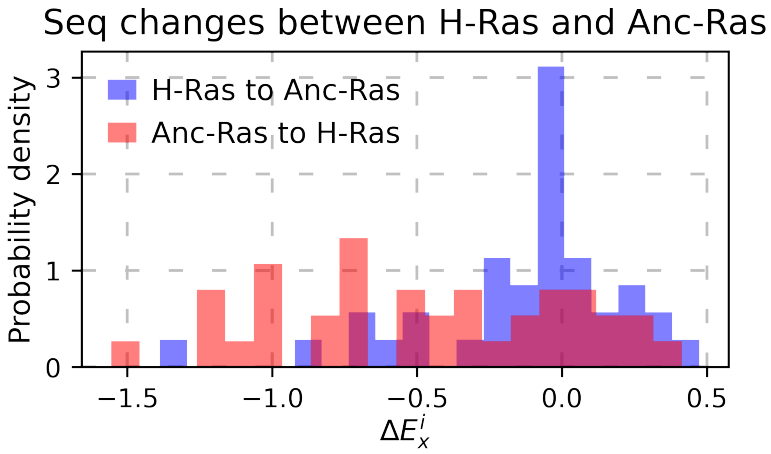
```
    x_label='H-Ras Unregulated',
    y_label='H-Ras Regulated',
)
```





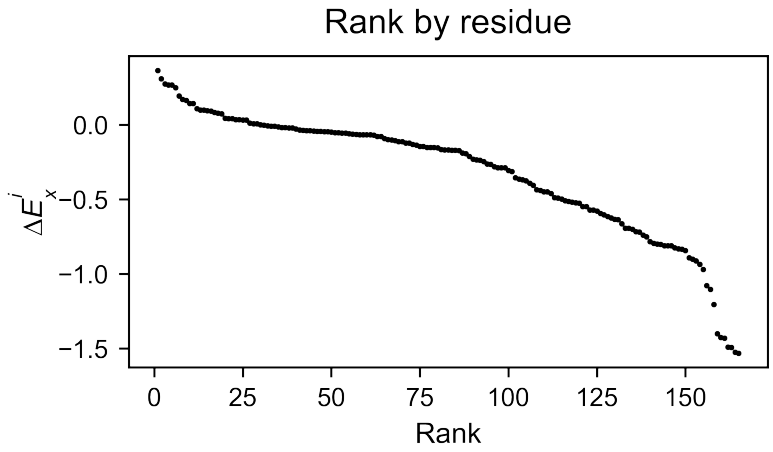
If you are comparing two homologs/paralogs, you can evaluate what happens when mutating every site that differs between the two proteins to the identity in the other second protein. (ie K4A and A4K)

```
# here map the residues that are different between the two proteins
map_sequence_changes = [(1, 1), (5, 5), (56, 56), (122, 123)]
                        #^ same residue             #^ the residue 122␣
↪of the first protein matches the 123 rd of the second protein
# ancestralras_rbd does not exist yet, so this cell wont run
hras_rbd.sequence_differences(ancestralras_rbd, map_sequence_changes)
```



The method `object.rank` sorts each mutation (or position) by its enrichment score.

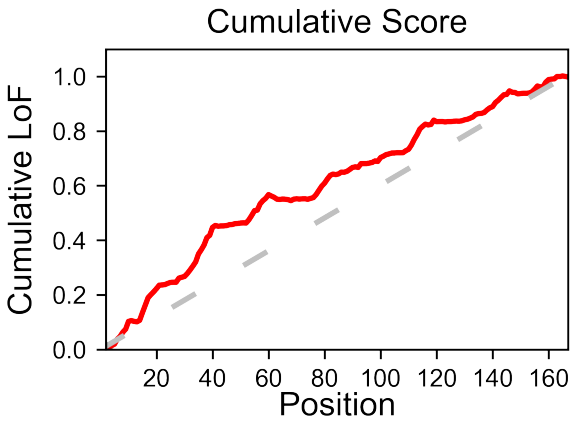
```
hras_rbd.rank(mode='pointmutant', outdf=True, title='Rank of mutations
↪')
```



Rank	Variant	Score
0	R149H	2.363
1	R128A	1.196
2	L159I	0.963
3	V160M	0.732
4	Q25G	0.682

The method `object.cumulative` draws a cumulative plot that sums the mean enrichment score of every position. This plot is useful to determine if the sensitivity to mutations is constant throughout the protein or not. In the example, we see that the cumulative function follows the x=y line, suggestion a homogeneous mutational tolerance.

```
# Cumulative plot
hras_rbd.cumulative(mode='all', title='Cumulative Score')
```



4.6.3 Bar and line charts

Classes reviewed in this section:

- *mutagenesis_visualization.main.bar_graphs.enrichment_bar.EnrichmentBar*
- *mutagenesis_visualization.main.bar_graphs.differential.Differential*
- *mutagenesis_visualization.main.bar_graphs.position_bar.PositionBar*
- *mutagenesis_visualization.main.bar_graphs.secondary.Secondary*

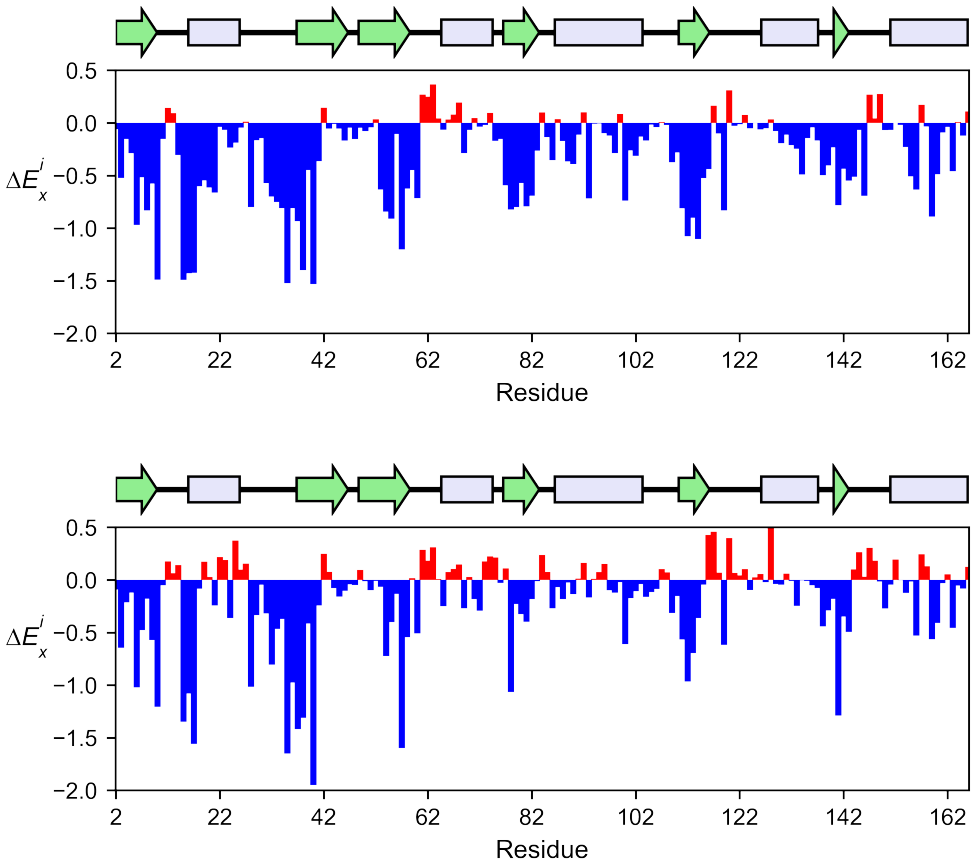
The method `object.enrichment_bar` will plot the mean enrichment score for every position on a bar chart. It will be colored blue for loss of function and red for gain of function. Additionally, setting the parameter `mode` to an amino acid (using the one letter code) will plot the enrichment for that particular amino acid along the protein. In this example, we are showing the mean enrichment scores (top) and an alanine scan (bottom)

```
# Plot a bar graph with the mean enrichment score
hras_rbd.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[-2, 0.5],
    title='',
)
```

(continues on next page)

(continued from previous page)

```
# Plot a bar graph with the alanine enrichment score
hras_rbd.enrichment_bar(
    figsize=[6, 2.5],
    mode='A',
    show_cartoon=True,
    yscale=[-2, 0.5],
    title='',
)
```



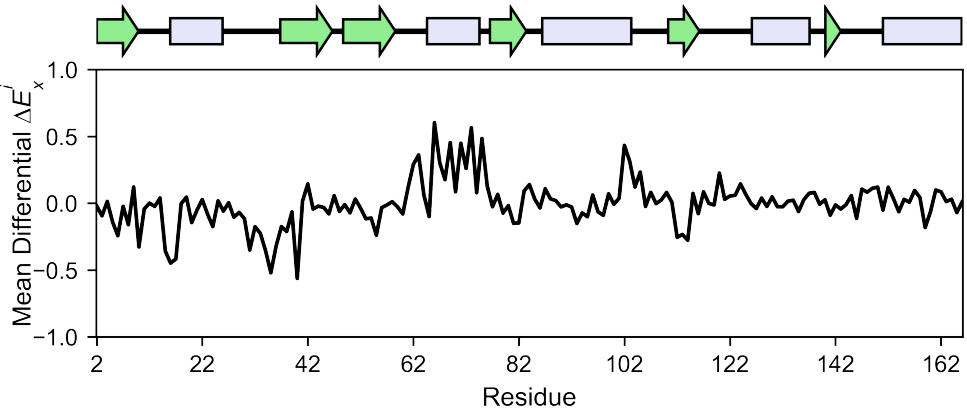
The mean differential effect between the two example datasets is displayed using the method `object.differential`. This plot is useful to compare either orthologs/paralogs or the same protein with different effectors, and determine which areas of the protein have a different sensitivity to mutations.

```
# Plot the difference between H-Ras unregulated and H-Ras regulated␣
↪datasets
# The subtraction is hras_RBD - hrasGAPGEF
hras_rbd.differential(
    hras_gapgef,
```

(continues on next page)

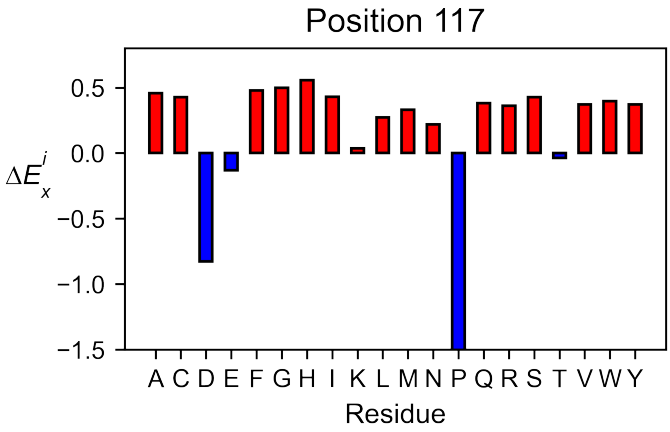
(continued from previous page)

```
    figsize=[6, 2.5],
    show_cartoon=True,
    yscale=[-1, 1],
    title='',
)
```



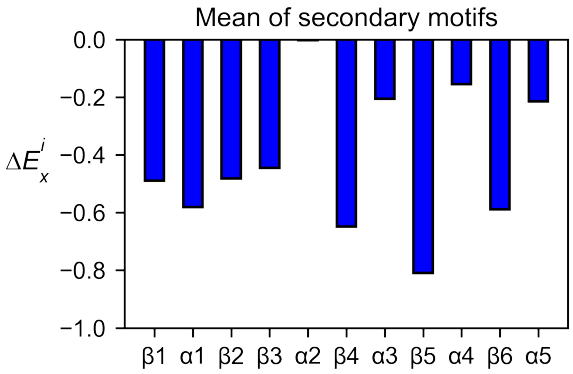
You can check the individual mutational profile of a residue by using `object.position_bar`.

```
# Create plot for position 117
hras_rbd.position_bar(
    position=117,
    yscale=(-1.5, 0.8),
    figsize=(3.5, 2),
    title='Position 117',
)
```



If you added the secondary structure as an attribute of the object, you can plot the mean enrichment score for each alpha and beta motif in the protein (`object.secondary_mean`).

```
# Graph bar of the mean of each secondary motif
hras_rbd.secondary_mean(
    yscale=[-1, 0],
    figsize=[3, 2],
    title='Mean of secondary motifs',
    output_file=None
)
```



4.6.4 Correlation, PCA and ROC AUC

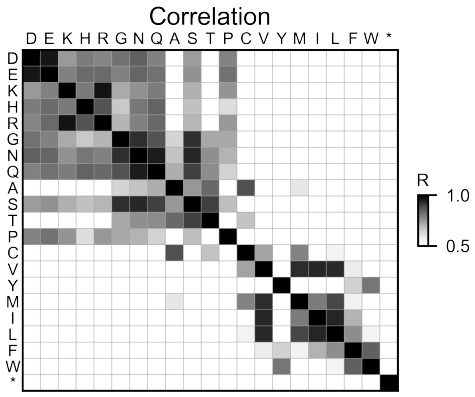
Classes reviewed in this section:

- *mutagenesis_visualization.main.pca_analysis.correlation.Correlation*
- *mutagenesis_visualization.main.pca_analysis.individual_correlation.IndividualCorrelation*
- *mutagenesis_visualization.main.pca_analysis.pca.PCA*
- *mutagenesis_visualization.main.other_stats.roc_analysis.ROC*

If you want to know more about PCA and ROC, watch the following StatQuest videos on youtube: PCA ROC and AUC

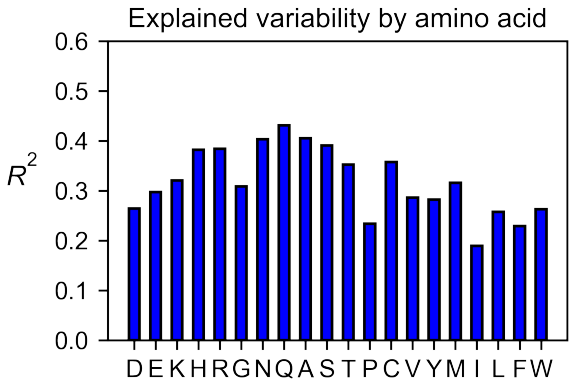
The correlation of amino acid substitution profiles can be calculated for each amino acid and graphed using `object.correlation`. In the example we observe that polar amino acids have high correlation between themselves but low correlation with hydrophobic amino acids.

```
# Correlation between amino acids
hras_rbd.correlation(
    colorbar_scale=[0.5, 1], title='Correlation'
)
```



The method `object.individual_correlation` will tell you how a single amino acid substitution profile (row of the heatmap) correlates to the rest of the dataset.

```
# Explained variability by amino acid
hras_rbd.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
    output_file=None
)
```



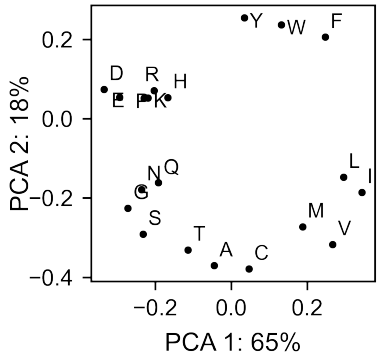
The package can perform principal component analysis (PCA) using the method `object.pca`. The parameter `mode` can be set to `aminoacid`, in which will cluster amino acids based on their similarity, `individual` in which will do the same for each individual residue and `secondary`, in which will cluster for each motif. By default, the first two dimensions will be plotted (0 and 1 in Python notation), but that can be changed by `dimensions` parameter.

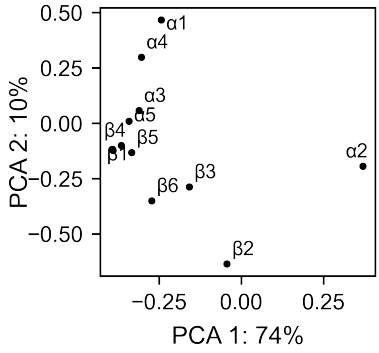
```
# PCA by amino acid substitution
hras_rbd.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
```

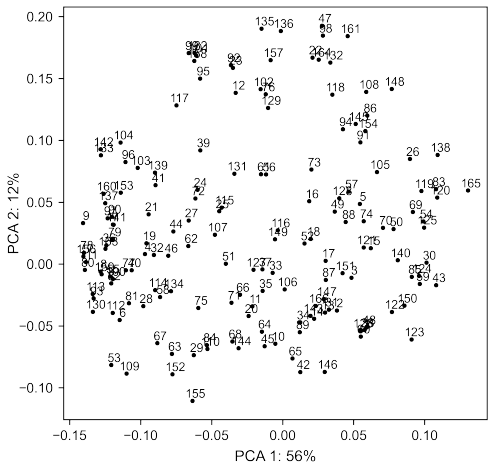
(continues on next page)

(continued from previous page)

```
    output_file=None
)

# PCA by secondary structure motif
hras_rbd.pca(
    title='',
    mode='secondary',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
    output_file=None
)

# PCA by each individual residue. Don't set adjustlabels = True unless
↪really big figsize
hras_rbd.pca(
    title='',
    mode='individual',
    dimensions=[0, 1],
    figsize=(5, 5),
    adjustlabels=False,
    output_file=None
)
```

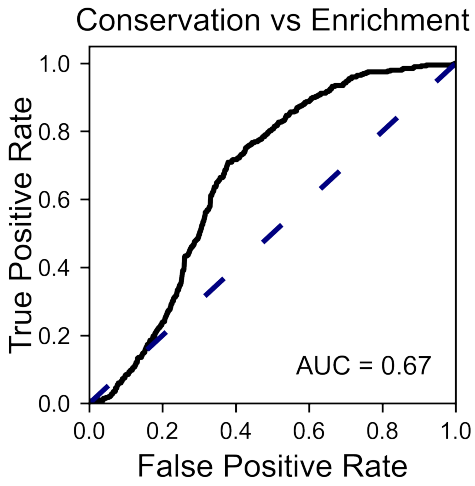






Another type of plot that can be done is a receiver operating characteristic (ROC) curve for classification. You will use the method `object.roc` and as an input you will pass a dataframe that contains the label for each variant.

```
#  Fake data
df_freq: DataFrame = DataFrame()
df_freq['Variant'] = hras_rbd.dataframes.df_notstopcodons[-1]['Variant
↪']
df_freq['Class'] = np.random.randint(2, size=len(df_freq))

# Plot ROC curve
hras_rbd.roc(
    df_freq[['Variant', 'Class']],
    title='ROC example',
)
```



4.6.5 Pymol

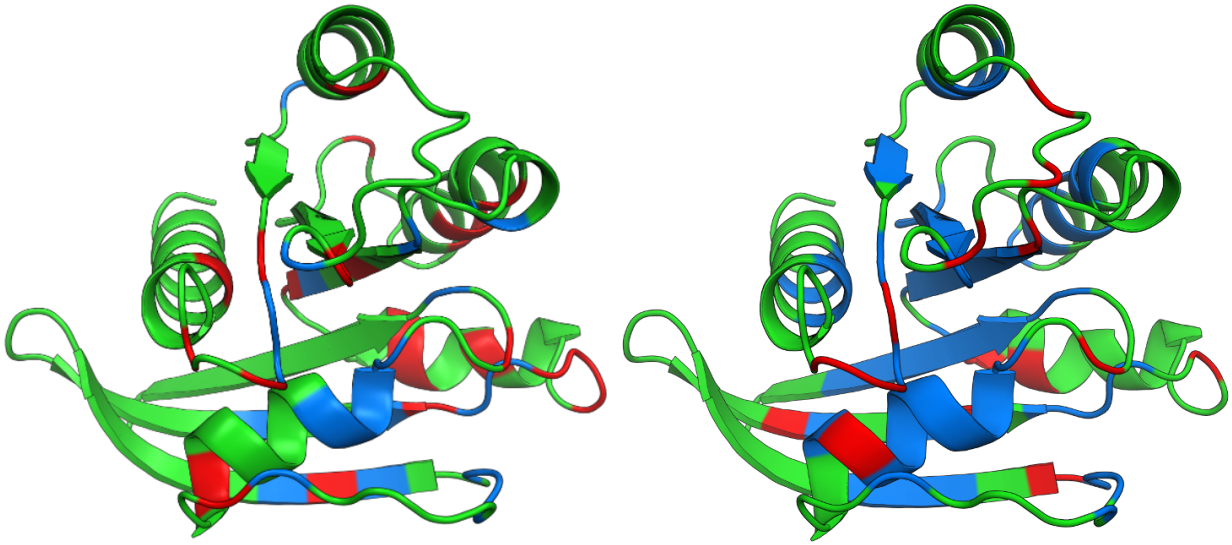
Class reviewed in this section:

- *mutagenesis_visualization.main.pymol.pymol.Pymol*

The data can be visualized on a Pymol object using `object.pymol`. It is important that not only Pymol is installed, but also on the same path as Python. You may have to manually install the ipymol API. See the Getting Started chapter for more information.

The parameter `pdb` will fetch the pdb that you want to use. Note that the protein chain needs to be specified (see example). Red for gain of function and blue for loss of function. `mode` lets you specifiy whether to plot the mean or an individual amino acid profile (left - Leucine, right - Aspartate).

```
# Start pymol and color residues. Cut offs are set with gof and lof␣
↪parameters.
hras_rbd.pymol(pdb='5p21_A', mode='mean', gof=0.2, lof=-0.5)

# Now check the mutational profile of Leucine (left image)
hras_rbd.pymol(pdb='5p21_A', mode='L', gof=0.2, lof=-0.5)

# Now check the mutational profile of Aspartate (right image)
hras_rbd.pymol(pdb='5p21_A', mode='D', gof=0.2, lof=-0.5)
```



Leucine scan

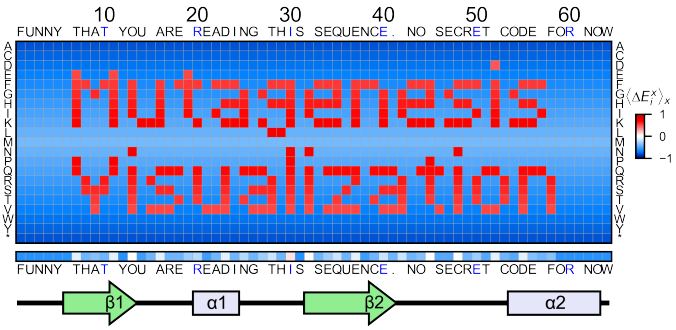
Aspartate scan

4.6.6 Art

The heatmap method can be used to generate artistic plots such as the one in the documentation overview. In here we show how that is done. On an Excel we have defined the color for each square

in the heatmap (also available with the package, see `logo.xlsx`). The first step is to import the excel file, and then we perform the same steps as in a normal dataset.

```
%matplotlib inline

from mutagenesis_visualization.main.classes.screen import Screen
from mutagenesis_visualization.main.utils.data_paths import PATH_LOGO
# Read excel file
usecols = 'A:BL'
#df_logo = pd.read_excel(path, 'logo', usecols=usecols, nrows=21)
#df_faded = pd.read_excel(path, 'logo_faded', usecols=usecols,␣
↪nrows=21)
df_logo = pd.read_excel(PATH_LOGO, 'logo_2', usecols=usecols, nrows=21)
df_faded = pd.read_excel(PATH_LOGO, 'logo_faded_2', usecols=usecols,␣
↪nrows=21)

# Combine two dataframes
df_mixed = df_logo * 1.2 - df_faded

# Aminoacids
aminoacids = list('ACDEFGHIKLMNPQRSTVWY*')

# Define protein sequence
sequence_logo = "FUNNY THAT YOU ARE READING THIS SEQUENCE. NO SECRET␣
↪CODE FOR NOW"

# Define secondary structure
secondary = [['L0'] * 5, ['β1'] * (9 - 1), ['L1'] * (15 - 9),
             ['α1'] * (25 - 20), ['L2'] * (32 - 25), ['β2'] * (42 -␣
↪32),
             ['L3'] * (50 - 42), ['α2'] * (58 - 50), ['L4'] * (70 -␣
↪58)]

# Create object
logo_obj = Screen(
    df_mixed, sequence_logo, aminoacids = aminoacids, start_position=1,
↪ fillna=0, secondary=secondary
)

# Create hetmap
logo_obj.heatmap(
    show_cartoon=True,
    title='',
    neworder_aminoacids='ACDEFGHIKLMNPQRSTVWY*',
)
```



4.6.7 Reference

4.7 Visualizing with plotly

In this section we will use Plotly to make interactive plots. Please let us know if you have suggestions for new figures that could be made with Plotly.

```
%matplotlib inline
import numpy as np
from mutagenesis_visualization.main.demo.demo_objects import␣
 ↪DemoObjects
from mutagenesis_visualization.main.utils.data_paths import PDB_5P21

DEMO_OBJECTS:DemoObjects = DemoObjects()
hras_rbd = DEMO_OBJECTS.hras_rbd
hras_gapgef = DEMO_OBJECTS.hras_gapgef
```

4.7.1 Heatmap

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.heatmap.HeatmapP*

Plot an interactive heatmap. Hopover individual pixels to get their characteristics.

```
hras_rbd.plotly_heatmap(
    title='H-Ras Heatmap',
    figsize=(6, 2.5),
)
```

4.7.2 Mean

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.enrichment_bar.EnrichmentBarP*

Analogous function to `object.mean` but rendered using plotly. Will plot the mean enrichment score for every position on a bar chart. It will be colored blue for loss of function and red for gain of function. Additionally, setting the parameter `mode` to an amino acid (using the one letter code) will plot the enrichment for that particular amino acid along the protein. In this example, we are showing the mean enrichment scores (top) and an alanine scan (bottom)

```
hras_rbd.plotly_enrichment_bar(
    title='Mean',
    figsize=(6, 2.5),
)

hras_rbd.plotly_enrichment_bar(
    title='A scan',
    mode='A',
    figsize=(6, 2.5),
)
```

4.7.3 Histogram

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.histogram.HistogramP*

Plot a histogram.

```
hras_rbd.plotly_histogram(
    title='Histogram',
    figsize=(3, 2.5),
)
```

4.7.4 Rank

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.rank.RankP*

Create an interactive rank figure that displays each mutant. The default mode is set to pointmutant to provide the ranking on the mutation level. You can download the plot as a png file by clicking

the camera icon which appears on the far left when our cursor is over the plot. You can export to an html file by giving a path to the variable `output_html`.

```
hras_rbd.plotly_rank(
    title='Rank of pointmutants',
)
```

Now display the rank of the positional mean.

```
hras_rbd.plotly_rank(mode='mean',title='Rank of positions')
```

4.7.5 Scatter

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.scatter.ScatterP*

If you have two datasets, you can create a scatter plot. The advantage of using plotly over matplotlib is that you can visually check each data point by hovoring your cursor over a point. By setting show_results = True, the OLS regression results will also be printed as output. The mode = 'pointmutant' is default which shows a comparison as mutation by mutation.

```
hras_rbd.plotly_scatter(
    hras_gapgef,
    show_results=False,
    title='Scatter Point Mutants',
    x_label='hras_rbd',
    y_label='hras_gapgef',
)
```

Now we just look at the positional average.

```
hras_rbd.plotly_scatter(
    hras_gapgef,
    mode='mean',
    title='Scatter Positional Average',
    x_label='hras_rbd',
    y_label='hras_gapgef',
)
```

4.7.6 3D scatter plot

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.scatter_3d.Scatter3D*

If there is an available PDB structure, you can input it and the software will plot a 3d plot of the C-alpha atoms, colored by their enrichment score.

The method `object.plotly_scatter_3d` will take as an input either a PDB file (`pdb_path=/path/to/file`) or the x,y,z coordinates (`df_coordinates`).

```
hras_rbd.plotly_scatter_3d(
    mode='mean',
    pdb_path=PDB_5P21,
    title='Scatter 3D',
    squared=False,
    x_label='x',
    y_label='y',
    z_label='z',
)
```

By setting up mode='V', we can evaluate the impact of valine substitutions. Mode can be set up to any residue. In this example, residues in the core are tolerant to valine substitutions.

```
hras_rbd.plotly_scatter_3d(
    mode='V',
    pdb_path=PDB_5P21,
    title='Scatter 3D - Valine substitution',
    squared=False,
    x_label='x',
    y_label='y',
    z_label='z',
)
```

When we set mode='D', the core of the protein turns completely blue.

```
hras_rbd.plotly_scatter_3d(
    mode='D',
    pdb_path=PDB_5P21,
    title='Scatter 3D - Aspartate substitution',
    squared=False,
    x_label='x',
    y_label='y',
    z_label='z',
)
```

By setting squared = True, we plot the distance to the center of the protein of each residue. In this example, we see that residues in the core of the protein are blue, indicating a sensitivity to mutations.

```
hras_rbd.plotly_scatter_3d(
    mode='mean',
```

(continues on next page)

(continued from previous page)

```
    pdb_path=PDB_5P21,
    title='Scatter 3D - Distance to center',
    squared=True,
    x_label='x',
    y_label='y',
    z_label='z',
)
```

4.7.7 PDB properties

Classes reviewed in this section:

- *mutagenesis_visualization.main.plotly.scatter_3d_pdb.Scatter3DPDB*

From the PDB, properties such as B-factor or SASA can be extracted. Using plotly we allow the user to have a 3-D scatter plot colored by the enrichment scores. You can additionally include other properties to include such as the conservation scores using the parameter `custom`.

```
# Plot 3-D SASA, log B-factor and Shannon Entropy
hras_rbd.plotly_scatter_3d_pdbprop(
    plot = ['Distance', 'SASA', 'log B-factor'],
    pdb_path=PDB_5P21,
    title='Scatter 3D - PDB properties',
)
```

4.8 Other datasets

Up to this moment, we have only shown how the package performs with our own dataset. The moment of truth is when we test our software with other people's datasets. In this section we have compiled saturation mutagenesis datasets found in the literature and we reproduce the analysis. Not only does the package works with other datasets, but also it allows to customize a wide range of parameters such as color maps, scales, etc. Furthermore, on top of testing the resilience of `mutagenesis_visualization`, we are providing extra examples on how to use this API.

```
%matplotlib inline
from typing import Dict, Union, List
from pandas.core.frame import DataFrame
import numpy as np
import pandas as pd
import matplotlib as plt
import copy
```

(continues on next page)

(continued from previous page)

```
from mutagenesis_visualization import Screen
from mutagenesis_visualization import load_demo_datasets
from mutagenesis_visualization import DemoObjects
from mutagenesis_visualization.main.utils.data_paths import PDB_1ERM,␣
↪PDB_1A5R, PDB_1ND4

DEMO_DATASETS: Dict[str, Union[np.array, DataFrame]] = load_demo_
↪datasets()
```

Function reviewed in this section:

- *mutagenesis_visualization.load_demo_datasets()*

4.8.1 Load objects

For simplicity, we also have added the option of loading those datasets into objects automatically There are 10 objects to load from other papers (hras_rbd, hras_gapgef, bla_obj, sumo_obj, mapk1_obj, ube2i_obj, tat_obj, rev_obj, asynuclein_obj, aph_obj, b11l5f_obj) and all the heatmaps from the Hidalgo et al. eLife 2022 paper (hras_166_gap, hras_166_rbd, hras_188_baf3, hras_180_gap, hras_180_rbd, kras_165_gap, kras_165_gapgef, kras_173_gapgef, kras_165_gef, kras_173_gef, kras_165_rbd, kras_173_gap, kras_173_rbd, hras_166_gapgef, krasq61l_173_gap, and krasq61l_173_rbd.).

4.8.2 Hidalgo et al. eLife 2022 paper

The figures of the Hidalgo et al. eLife 2022 paper were created using mutagenesis-visualization. To retrieve the objects, just instantiate the class `DemoObjects` and then access each of the datasets.

```
# instantiate
demo_objects = DemoObjects()

# access the datasets
demo_objects.hras_188_baf3.heatmap()
```

4.8.3 Beta Lactamase

Create object

```
#https://www.uniprot.org/uniprot/P62593#sequences

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids: List[str] = list(DEMO_DATASETS['df_bla'].index)
neworder_aminoacids: List[str]  = list('DEKHRGNQASTPCVYMILFW')

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_bla'].columns[0]

# Define sequence. If you dont know the start of the sequence, just␣
↪add X's
sequence_bla_x =
↪'MSIQHFRVALIPFFAAFCLPVFAHPETLVKVKDAEDQLGARVGYIELDLNSGKILESFRP' +
↪'EERFPMMSTFKVLLCGAVLSRVDAGQEQLGRRIHYSQNDLVEYSPVTEKHLTDGMTVREL' +
↪'CSAAITMSDNTAANLLLTTIGGPKELTAFLHNMGDHVTRLDRWEPELNEAIPNDERDTTM' +
↪'PAAMATTLRKLLTGELLTLASRQQLIDWMEADKVAGPLLRSALPAGWFIADKSGAGERGS' +
↪'RGIIAALGPDGKPSRIVVIYTTGSQATMDERNRQIAEIGASLIKHW'

# Define secondary structure
secondary_bla = [['L0'] * 23, ['α1'] * (38 - 23), ['L1'] * 2, ['β1'] *␣
↪(48 - 40),
                 ['L2'] * 5, ['β2'] * (57 - 53), ['L3'] * (68 - 57),
↪ ['α2'] * (84 - 68),
                 ['L4'] * (95 - 84), ['α3'] * (100 - 95), ['L5'] *␣
↪(103 - 100),
                 ['α4'] * (110 - 103), ['L6'] * (116 - 110), ['α5']␣
↪* (140 - 116),
                 ['L7'] * (1), ['α6'] * (153 - 141), ['L8'] * (164 -
↪ 153),
                 ['α7'] * (169 - 164), ['L9'] * (179 - 169), ['α8']␣
↪* (194 - 179), ['L10'] *
                 3, ['α9'] * (210 - 197), ['L11'] * (227 - 210), [
↪'β3'] * (235 - 227),
                 ['L12'] * (240 - 235), ['β4'] * (249 - 240), ['L13
↪'] * (254 - 249),
                 ['β5'] * (262 - 254), ['L14'] * (266 - 262), ['α10
↪'] * (286 - 266)]

bla_obj: Screen = Screen(
    DEMO_DATASETS['df_bla'], sequence_bla_x, aminoacids, start_
↪position, 0, secondary_bla
)
```

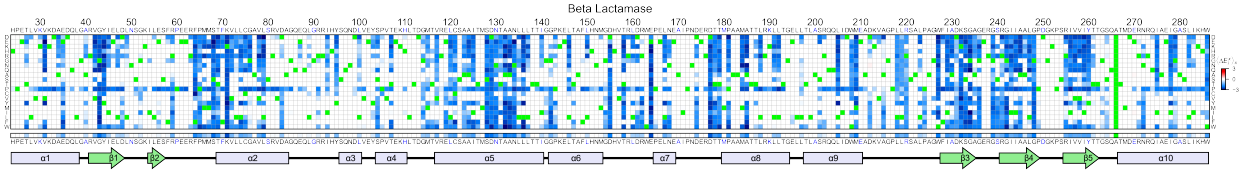
2D Plots

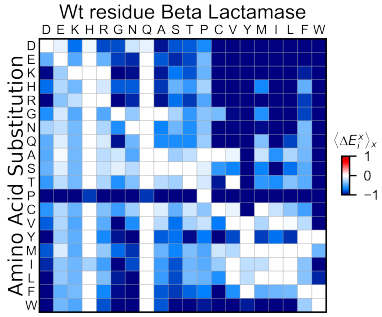
```
# Create full heatmap
bla_obj.heatmap(
    colorbar_scale=(-3, 3),
    neworder_aminoacids=neworder_aminoacids,
    title='Beta Lactamase',
    show_cartoon=True,
)

# Miniheatmap
bla_obj.miniheatmap(
    title='Wt residue Beta Lactamase',
    neworder_aminoacids=neworder_aminoacids,
)

# Positional mean
bla_obj.enrichment_bar(
    figsize=[10, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[-3, 0.25],
    title='',
)

# Kernel
bla_obj.kernel(
    histogram=True, title='Beta Lactamase', xscale=[-4, 1]
)

# Graph bar of the mean of each secondary motif
bla_obj.secondary_mean(
    yscale=[-1.5, 0],
    figsize=[5, 2],
    title='Mean of secondary motifs',
)

# Correlation between amino acids
bla_obj.correlation(
    colorbar_scale=[0.5, 1],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
bla_obj.individual_correlation(
```

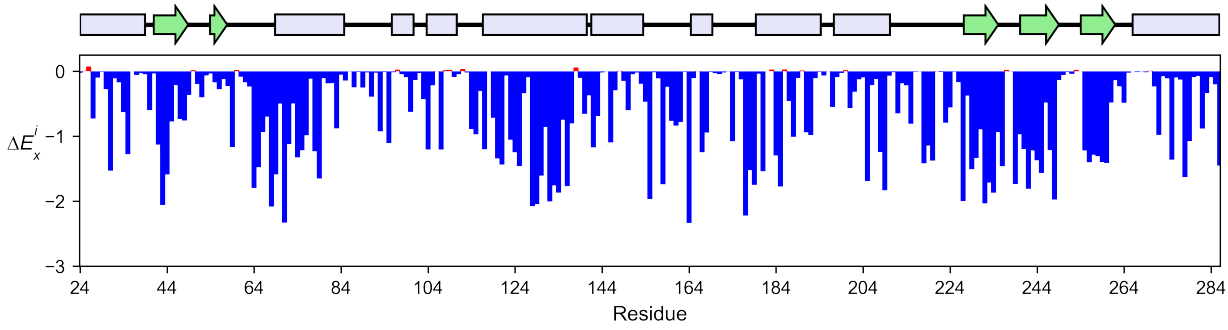
(continues on next page)

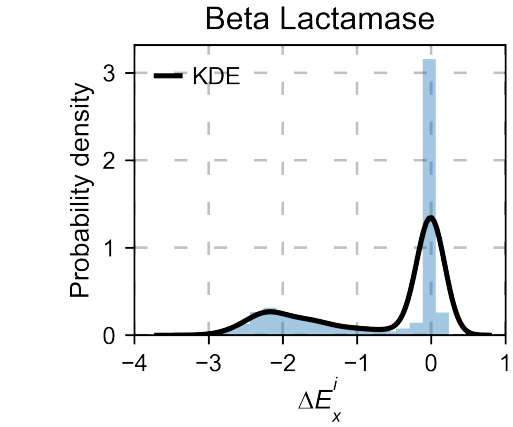
(continued from previous page)

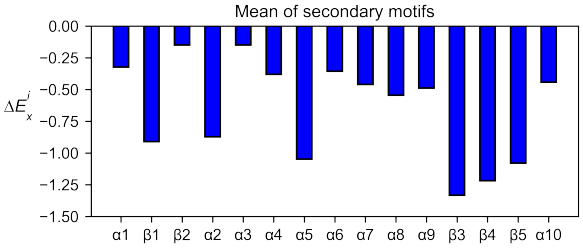
```
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
bla_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)

# PCA by secondary structure motif
bla_obj.pca(
    title='',
    mode='secondary',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

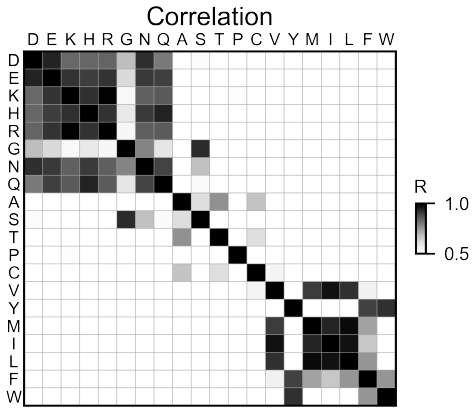


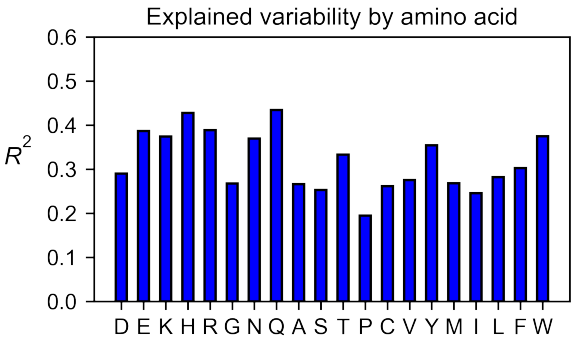


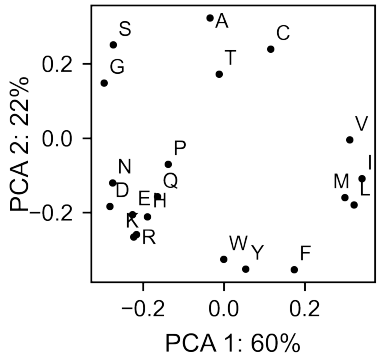


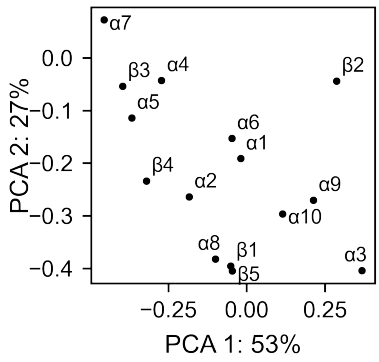

Beta Lactamase
KDE
Probability density
ΔE_x^i


Mean of secondary motifs
ΔE_x^i
α1 β1 β2 α2 α3 α4 α5 α6 α7 α8 α9 β3 β4 β5 α10


Correlation
D E K H R G N Q A S T P C V Y M I L F W
R
1.0
0.5


Explained variability by amino acid
R^2
D E K H R G N Q A S T P C V Y M I L F W

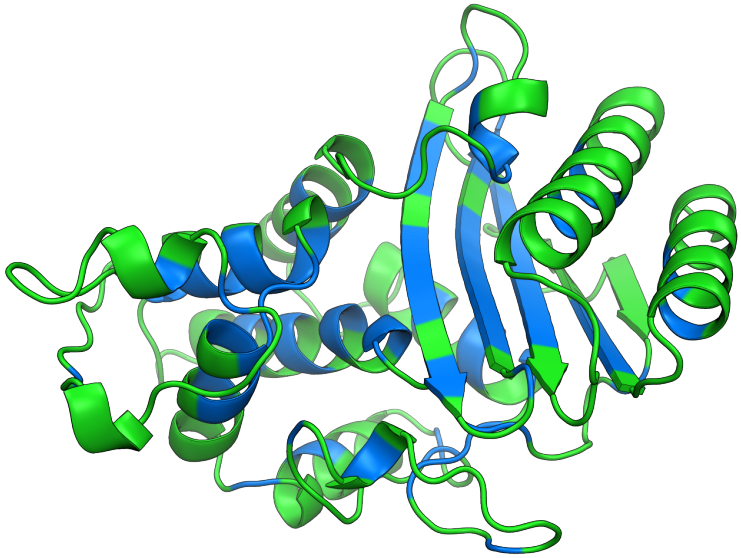




3D Plots

```
# Plot 3-D plot
bla_obj.plotly_scatter_3d(
    mode='mean',
    pdb_path=PDB_1ERM,
    position_correction=2,
    title='Scatter 3D',
    squared=False,
    x_label='x',
    y_label='y',
    z_label='z',
)

# Plot 3-D of distance to center of protein, SASA and B-factor
bla_obj.plotly_scatter_3d_pdbprop(
    plot=['Distance', 'SASA', 'log B-factor'],
    position_correction=2,
    pdb_path=PDB_1ERM,
    title='Scatter 3D - PDB properties',
)
```

```
# Start pymol and color residues. Cut offs are set with gof and lof␣
↪parameters.
bla_obj.pymol(
    pdb=PDB_1ERM, mode='mean', gof=0.2, lof=-1, position_correction=2
)
```



4.8.4 Sumo1

Create object

```
#https://doi.org/10.15252/msb.20177908

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_sumo1'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_sumo1'].columns[0]

# Full sequence
sequence_sumo1 =
↪'MSDQEAKPSTEDLGDKKEGEYIKLKVIGQDSSEIHFKVKMTTHLKKLKESYCQRQGVPMN' +
↪'SLRFLFEGQRIADNHTPKELGMEEEDVIEVYQEQTGGHSTV'

# Define secondary structure
secondary_sumo1 = [['L0'] * (20), ['β1'] * (28 - 20), ['L1'] * 3, ['β2
↪'] * (39 - 31),
```

(continues on next page)

(continued from previous page)

```
                    ['L2'] * 4, ['α1'] * (55 - 43),
                    ['L3'] * (6), ['β3'] * (65 - 61), ['L4'] * (75 -␣
↪65), ['α2'] * (80 - 75),
                    ['L5'] * (85 - 80), ['β4'] * (92 - 85), ['L6'] *␣
↪(101 - 92)]

sumo_obj: Screen = Screen(
    DEMO_DATASETS['df_sumo1'], sequence_sumo1, aminoacids, start_
↪position, 1,
    secondary_sumo1
)
```

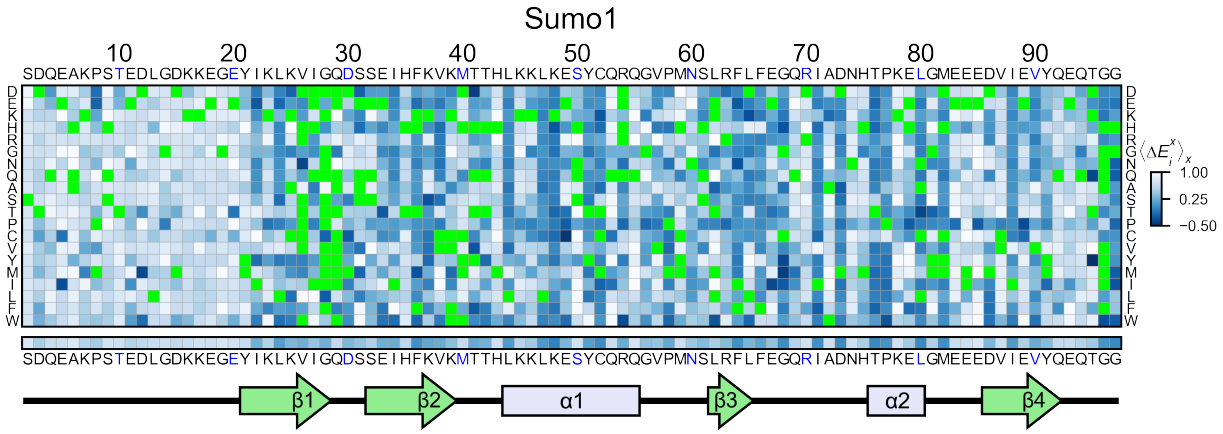
2D Plots

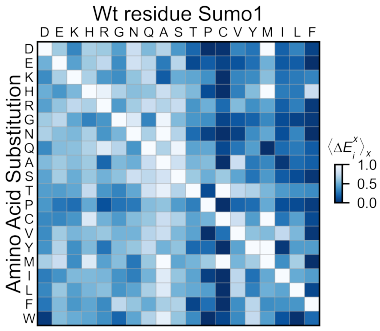
```
# You can use your own colormap or import it from matplotlib
colormap = copy.copy((plt.cm.get_cmap('Blues_r')))

# Create full heatmap
sumo_obj.heatmap(
    colorbar_scale=(-0.5, 1),
    neworder_aminoacids=neworder_aminoacids,
    title='Sumo1',
    colormap=colormap,
    show_cartoon=True,
)

# Miniheatmap
sumo_obj.miniheatmap(
    colorbar_scale=(0, 1),
    title='Wt residue Sumo1',
    neworder_aminoacids=neworder_aminoacids,
    colormap=colormap,
)

# Positional mean
sumo_obj.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[0, 1],
    title='',
)

# Kernel
```

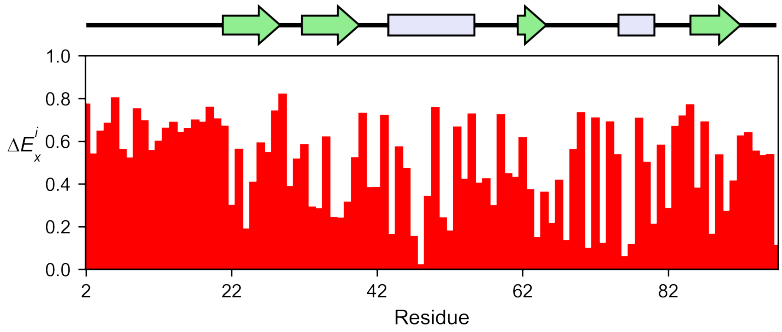
(continues on next page)

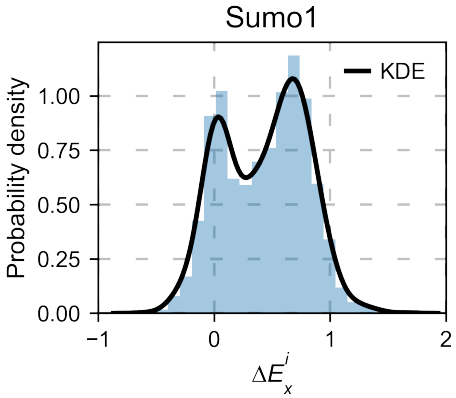
(continued from previous page)

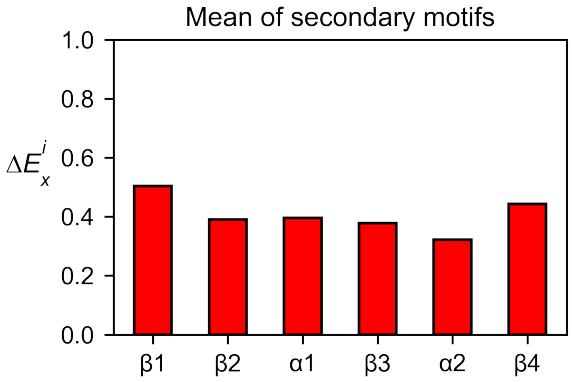
```
sumo_obj.kernel(histogram=True, title='Sumo1', xscale=[-1, 2], output_
↪file=None)

# Graph bar of the mean of each secondary motif
sumo_obj.secondary_mean(
    yscale=[0, 1],
    figsize=[2, 2],
    title='Mean of secondary motifs',
)

# Correlation between amino acids
sumo_obj.correlation(
    colorbar_scale=[0.25, 0.75],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
sumo_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
sumo_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)

# PCA by secondary structure motif
sumo_obj.pca(
    title='',
    mode='secondary',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

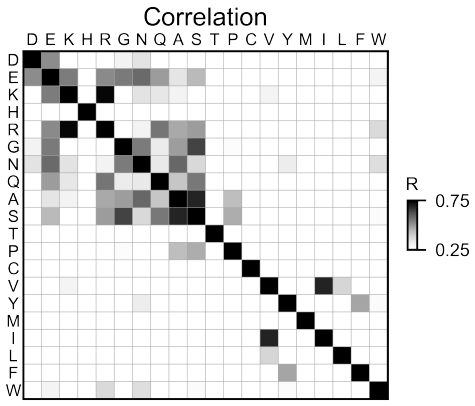

Sumo1
10 20 30 40 50 60 70 80 90
SDQEAKPSTEDLGDKKEGEYIKLKVIGQDSSEIHFKVKMTTHLKKLKESYCQRQGVPMNSLRFLFEGQRIADNHTPKELGMEEEDVIEVYQEQTGG
1.00
0.25
−0.50
β1 β2 α1 β3 α2 β4

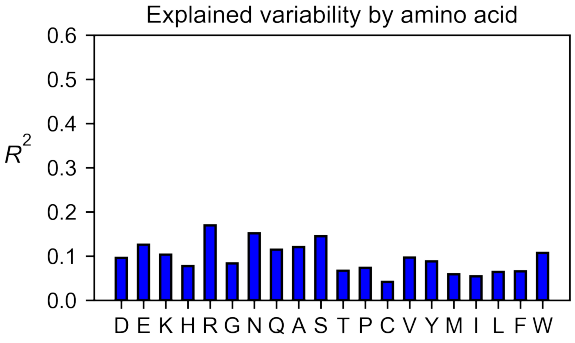

Wt residue Sumo1
Amino Acid Substitution
1.0
0.5
0.0

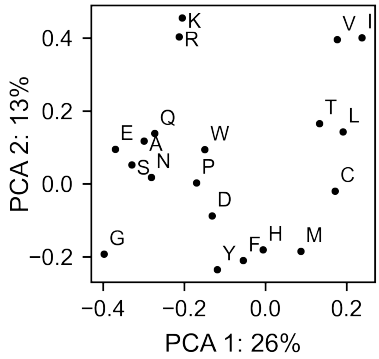

1.0
0.8
0.6
0.4
0.2
0.0
2
22
42
62
82
Residue

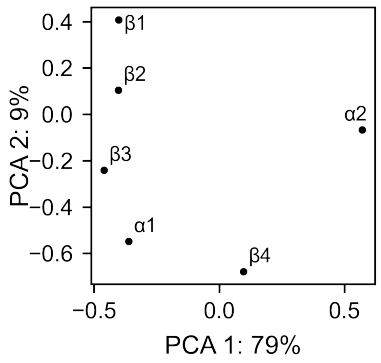

Sumo1
KDE
Probability density
1.00
0.75
0.50
0.25
0.00
−1
0
1
2


Mean of secondary motifs
ΔE_x^i
1.0
0.8
0.6
0.4
0.2
0.0
β1
β2
α1
β3
α2
β4

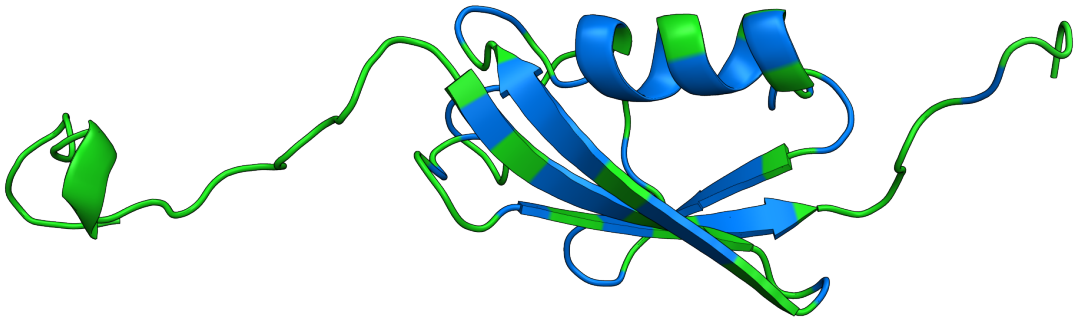

Correlation
D E K H R G N Q A S T P C V Y M I L F W
R
0.75
0.25


Explained variability by amino acid
R^2
0.6
0.5
0.4
0.3
0.2
0.1
0.0
D E K H R G N Q A S T P C V Y M I L F W


PCA 2: 13%
PCA 1: 26%
0.4
0.2
0.0
−0.2
−0.4
−0.2
0.0
0.2
K
R
V
I
Q
E
A
W
T
L
S
N
P
C
D
G
Y
F
H
M




```
# Open pymol and color the sumo structure
sumo_obj.pymol(pdb=PDB_1A5R, mode='mean', gof=1, lof=0.5)
```



4.8.5 MAPK1

Create object

```
# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_mapk1'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_mapk1'].columns[0]

# Full sequence
```

(continues on next page)

(continued from previous page)

```
sequence_mapk1_x =
↪'MAAAAAAGAGPEMVRGQVFDVGPRYTNLSYIGEGAYGMVCSAYDNVNKVRVAIK' +
↪'KISPFEHQTYCQRTLREIKILLRFRHENIIGINDIIRAPTIEQMKDVYIVQDLMETDLYKLLKTQ'␣
↪+ 'HLSNDHICYFLYQILRGLKYIHSANVLHRDLKPSNLLLNTTCDLKICDFGLARVADPDHDHTGFL
↪' +
↪'TEYVATRWYRAPEIMLNSKGYTKSIDIWSVGCILAEMLSNRPIFPGKHYLDQLNHILGILGSPSQ'␣
↪+ 'EDLNCIINLKARNYLLSLPHKNKVPWNRLFPNADSKALDLLDKMLTFNPHKRIEVEQALAHPYLE
↪' + 'QYYDPSDEPIAEAPFKFDMELDDLPKEKLKELIFEETARFQPGYRS'

# Create objects
mapk1_obj: Screen = Screen(DEMO_DATASETS['df_mapk1'], sequence_mapk1_x,
↪ aminoacids, start_position, 0)
```

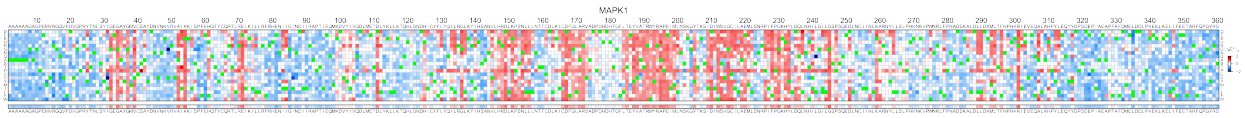
2D Plots

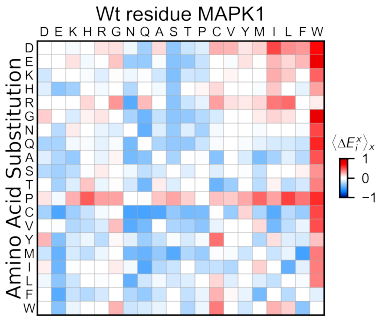
```
# Create full heatmap
mapk1_obj.heatmap(
    colorbar_scale=(-2, 2),
    neworder_aminoacids=neworder_aminoacids,
    title='MAPK1',
    show_cartoon=False,
)

# Miniheatmap
mapk1_obj.miniheatmap(
    title='Wt residue MAPK1',
    neworder_aminoacids=neworder_aminoacids,
)

# Positional mean
mapk1_obj.enrichment_bar(
    figsize=[10, 2.5],
    mode='mean',
    show_cartoon=False,
    yscale=[-1, 1],
    title='',
)

# Kernel
mapk1_obj.kernel(
    histogram=True, title='MAPK1', xscale=[-2, 2], output_file=None
)

# Correlation between amino acids
```

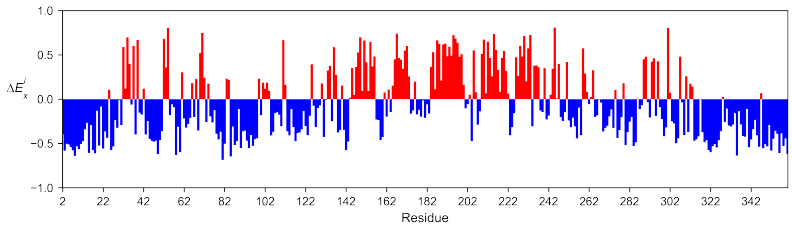
(continues on next page)

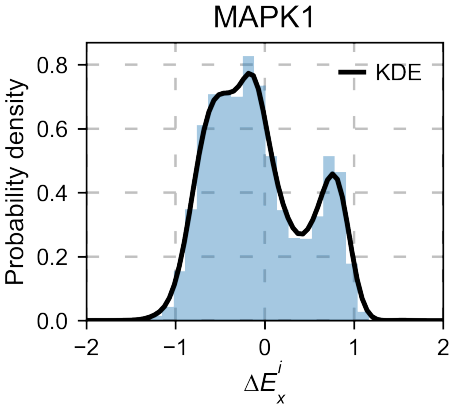
(continued from previous page)

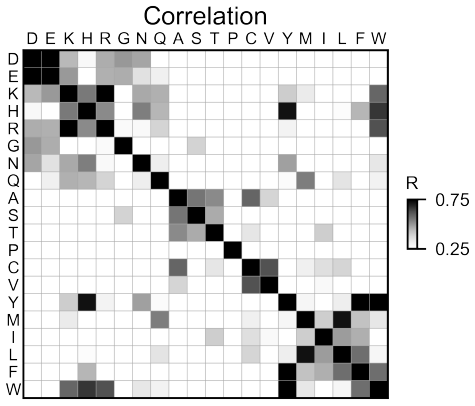
```
mapk1_obj.correlation(
    colorbar_scale=[0.25, 0.75],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
mapk1_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
mapk1_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

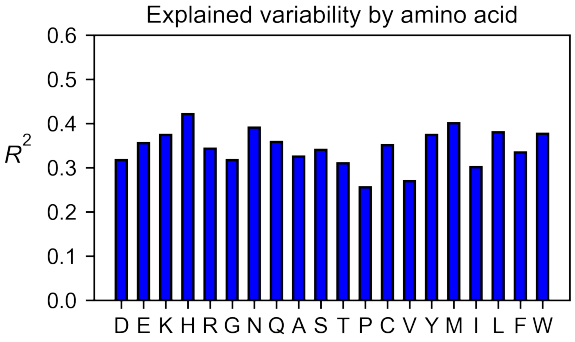


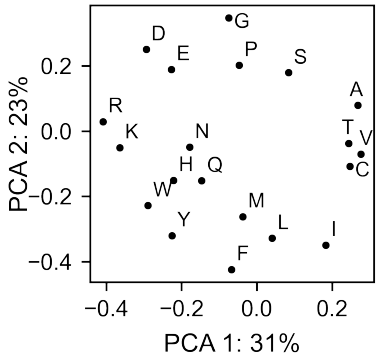





MAPK1
KDE
Probability density
0.0
0.2
0.4
0.6
0.8
−2
−1
0
1
2
ΔE_x^j


Correlation
D E K H R G N Q A S T P C V Y M I L F W
R
0.75
0.25


Explained variability by amino acid
R^2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
D E K H R G N Q A S T P C V Y M I L F W


PCA 2: 23%
PCA 1: 31%
−0.4
−0.2
0.0
0.2
D
E
P
G
S
A
R
K
N
T
V
C
H
Q
W
M
Y
L
I
F

4.8.6 UBE2I

Create object

```
# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_ube2i'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_ube2i'].columns[0]

# Full sequence
sequence_ube2i_x =
↪'MSGIALSRLAQERKAWRKDHPFGFVAVPTKNPDGTMNLMNWECAIPGKKGTP' +
↪'WEGGLFKLRMLFKDDYPSSPPKCKFEPPLFHPNVYPSGTVCLSILEEDKDWRPAITIKQ' +
↪'ILLGIQELLNEPNIQDPAQAEAYTIYCQNRVEYEKRVRAQAKKFAPS'

# Define secondary structure
secondary_ube2i = [['α1'] * (20 - 1), ['L1'] * (24 - 20), ['β1'] * (30␣
↪- 24), ['L2'] * 5,
                   ['β2'] * (46 - 35), ['L3'] * (56 - 46), ['β3'] *␣
↪(63 - 56),
                   ['L4'] * (73 - 63), ['β4'] * (77 - 73), ['L5'] *␣
↪(93 - 77),
                   ['α2'] * (98 - 93), ['L6'] * (107 - 98), ['α3'] *␣
↪(122 - 107),
                   ['L7'] * (129 - 122), ['α4'] * (155 - 129), ['L8']␣
↪* (160 - 155)]

# Create objects
ube2i_obj: Screen = Screen(
    DEMO_DATASETS['df_ube2i'], sequence_ube2i_x, aminoacids, start_
↪position, 1,
    secondary_ube2i
        )
```

2D Plots

```
colormap = copy.copy((plt.cm.get_cmap('Blues_r')))

# Create full heatmap
ube2i_obj.heatmap(
    colorbar_scale=(0, 1),
    neworder_aminoacids=neworder_aminoacids,
```

(continues on next page)

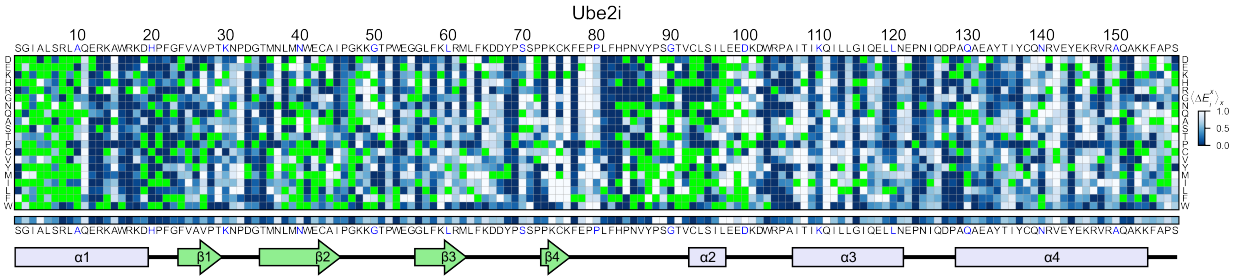
(continued from previous page)

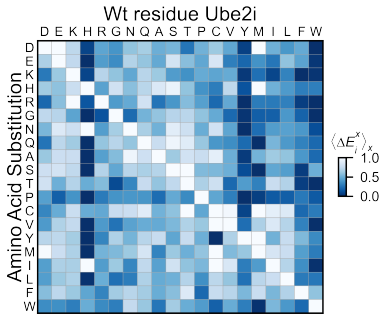
```
    title='Ube2i',
    colormap=colormap,
    show_cartoon=True,
)

# Miniheatmap
ube2i_obj.miniheatmap(
    colorbar_scale=(0, 1),
    title='Wt residue Ube2i',
    neworder_aminoacids=neworder_aminoacids,
    colormap=colormap,
)

# Positional mean
ube2i_obj.enrichment_bar(
    figsize=[10, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[0, 2],
    title='',
)

# Kernel
ube2i_obj.kernel(
    histogram=True, title='Ube2i', xscale=[-1, 2], output_file=None
)

# Graph bar of the mean of each secondary motif
ube2i_obj.secondary_mean(
    yscale=[0, 2],
    figsize=[3, 2],
    title='Mean of secondary motifs',
)

# Correlation between amino acids
ube2i_obj.correlation(
    colorbar_scale=[0.25, 0.75],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
ube2i_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
```

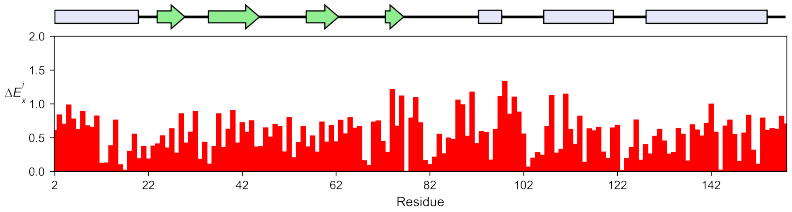
(continues on next page)

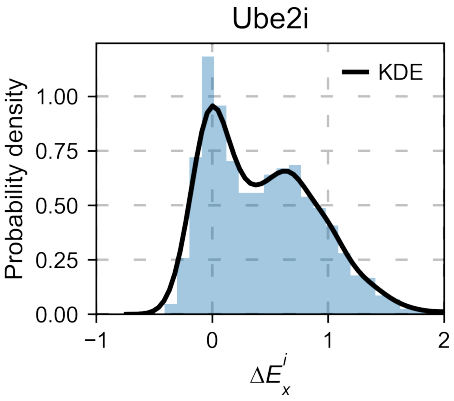
(continued from previous page)

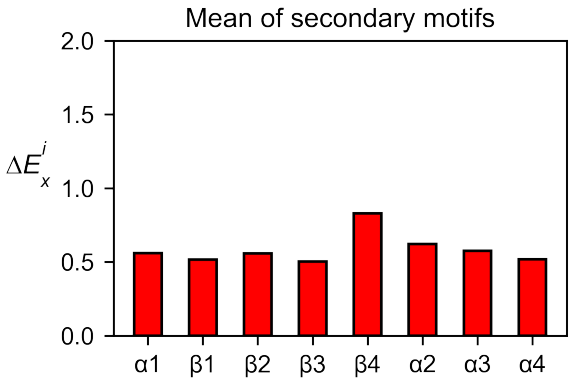
```
)

# PCA by amino acid substitution
ube2i_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)

# PCA by secondary structure motif
ube2i_obj.pca(
    title='',
    mode='secondary',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

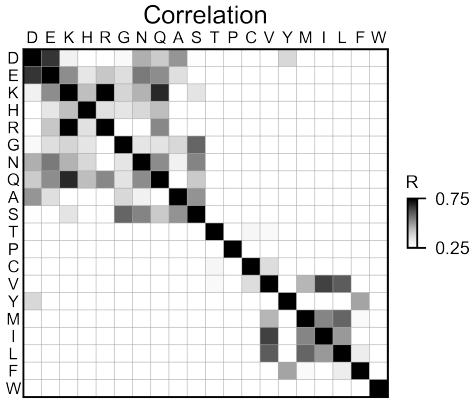


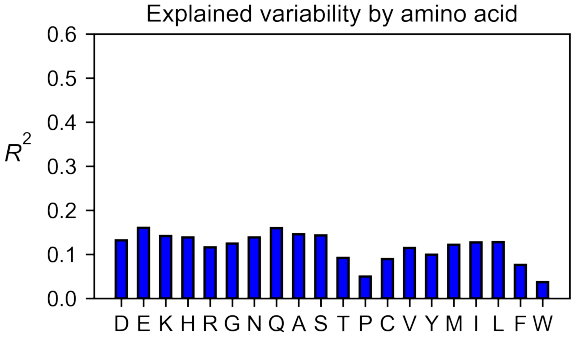


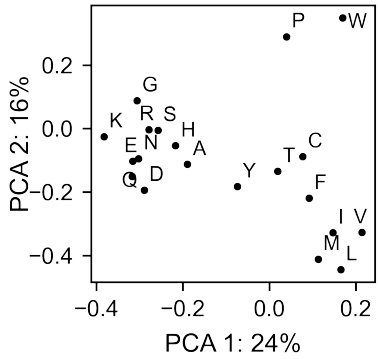


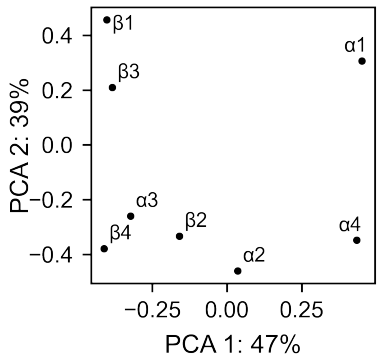

Ube2i
KDE
Probability density
1.00
0.75
0.50
0.25
0.00
−1
0
1
2
ΔE_x^j


Mean of secondary motifs
2.0
1.5
1.0
0.5
0.0
ΔE_x^j
α1
β1
β2
β3
β4
α2
α3
α4


Correlation
D E K H R G N Q A S T P C V Y M I L F W
R
0.75
0.25


Explained variability by amino acid
0.6
0.5
0.4
0.3
0.2
0.1
0.0
R^2
D E K H R G N Q A S T P C V Y M I L F W





4.8.7 TAT

Create object

```
#https://doi.org/10.1016/j.cell.2016.11.031

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_tat'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mutated, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_tat'].columns[0]

# Full sequence
sequence_tat =
↪'MEPVDPRLEPWKHPGSQPKTACTNCYCKKCCFHCQVCFITKALGISYGRKKRRQRRRAHQ' +
↪'NSQTHQASLSKQPTSQPRGDPTGPKE'

# Define secondary structure
secondary_tat = [['L1'] * (8), ['α1'] * (13 - 8), ['L2'] * (28 - 14), [
↪'α2'] * (41 - 28),
                 ['L3'] * (90 - 41)]
```

(continues on next page)

(continued from previous page)

```
tat_obj: Screen = Screen(
    DEMO_DATASETS['df_tat'], sequence_tat, aminoacids, start_position,␣
↪0, secondary_tat
)
```

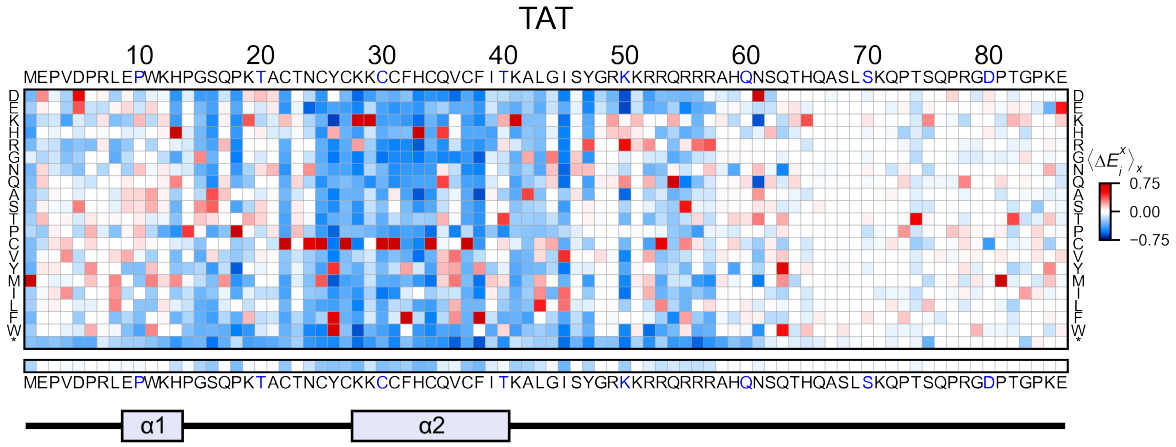
2D Plots

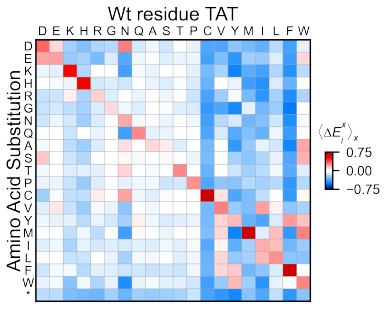
```
# Create full heatmap
tat_obj.heatmap(
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids,
    title='TAT',
    show_cartoon=True,

)

# Miniheatmap
tat_obj.miniheatmap(
    title='Wt residue TAT',
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids,

)

# Positional mean
tat_obj.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[-0.5, 0.25],
    title='',

)

# Kernel
tat_obj.kernel(histogram=True, title='TAT', xscale=[-1, 1], output_
↪file=None)

# Correlation between amino acids
tat_obj.correlation(
    colorbar_scale=[0.25, 1],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
```

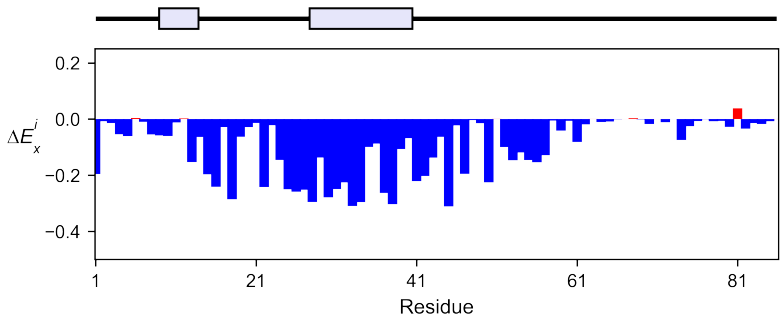
(continues on next page)

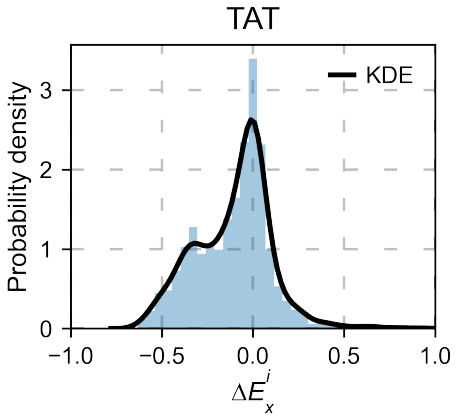
(continued from previous page)

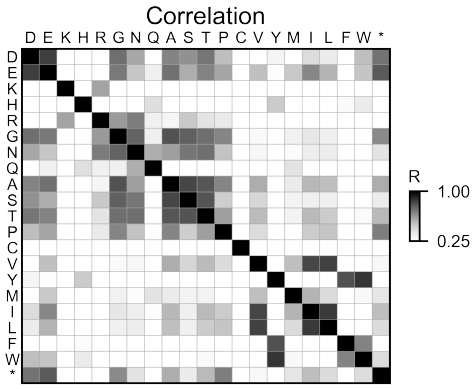
```
)

# Explained variability by amino acid
tat_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',

)

# PCA by amino acid substitution
tat_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,

)
```

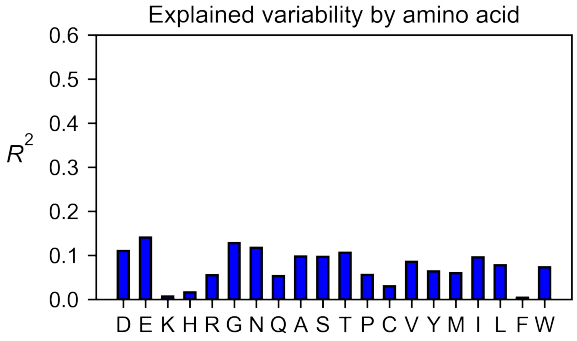


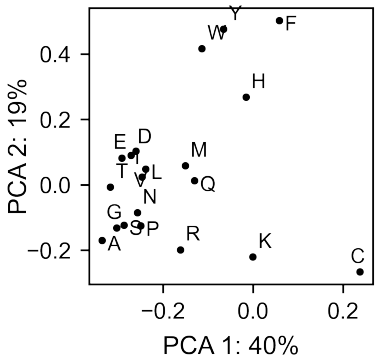



0.2
0.0
−0.2
−0.4
ΔE_x^i
1
21
41
61
81
Residue


TAT
KDE
Probability density
0
1
2
3
−1.0
−0.5
0.0
0.5
1.0
ΔE_x^i


Correlation
D E K H R G N Q A S T P C V Y M I L F W *
R
1.00
0.25


Explained variability by amino acid
R^2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
D E K H R G N Q A S T P C V Y M I L F W



4.8.8 REV

Create object

```
#https://doi.org/10.1016/j.cell.2016.11.031
#https://www.uniprot.org/uniprot/P69718

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_rev'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mureved, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_rev'].columns[0]

# Full sequence
sequence_rev =
↪'MAGRSGDSDEDLLKAVRLIKFLYQSNPPPNPEGTRQARRNRRRRWRERQRQIHSISERIL' +
↪'STYLGRSAEPVPLQLPPLERLTLDCNEDCGTSGTQGVGSPQILVESPTILESGAKE'

# Define secondary structure
secondary_rev = [['L1'] * (8), ['α1'] * (25 - 8), ['L2'] * (33 - 25), [
↪'α2'] * (68 - 33),
                 ['L3'] * (116 - 41)]

rev_obj: Screen = Screen(
    DEMO_DATASETS['df_rev'], sequence_rev, aminoacids, start_position,␣
↪0, secondary_rev
)
```

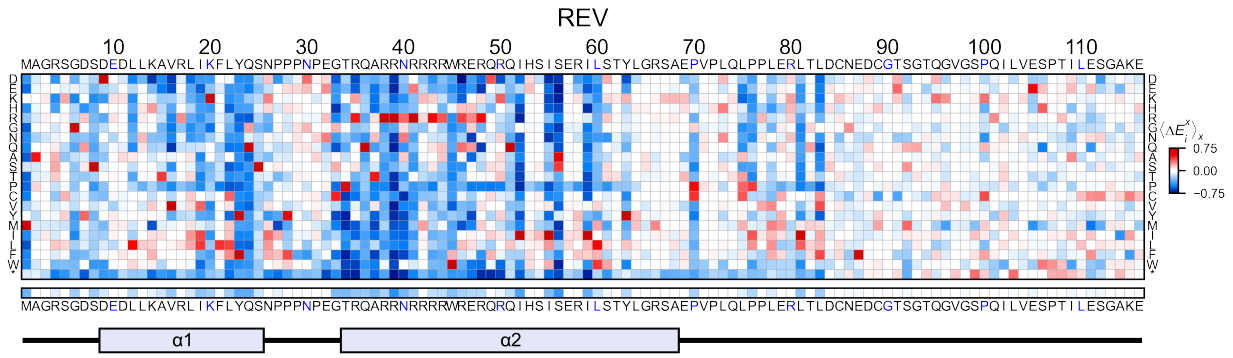
2D Plots

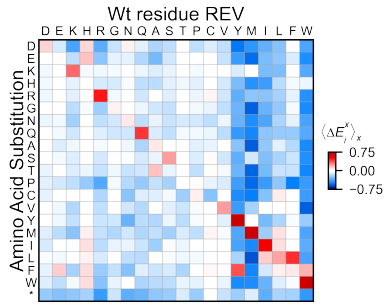
```
# Create full heatmap
rev_obj.heatmap(
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids+["*"],
    title='REV',
    show_cartoon=True,
)

# Miniheatmap
rev_obj.miniheatmap(
    title='Wt residue REV',
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids+["*"],
)

# Positional mean
rev_obj.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[-0.5, 0.25],
    title='',
)

# Kernel
rev_obj.kernel(histogram=True, title='REV', xscale=[-1, 1], output_
↪file=None)

# Correlation between amino acids
rev_obj.correlation(
    colorbar_scale=[0.25, 1],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
rev_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
rev_obj.pca(
    title='',
```

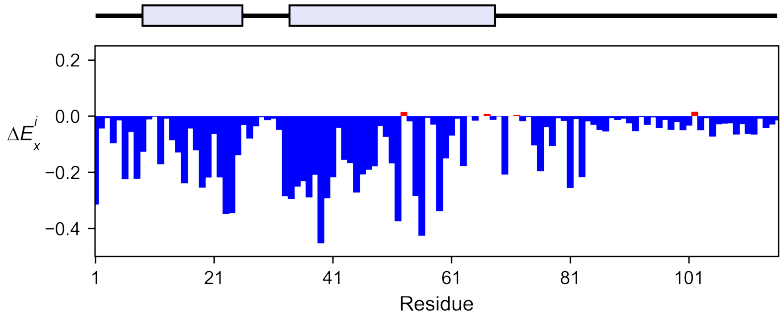
(continues on next page)

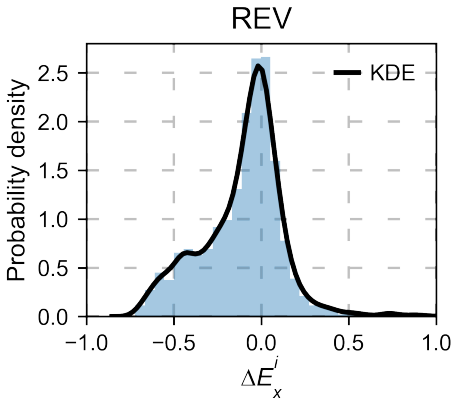
(continued from previous page)

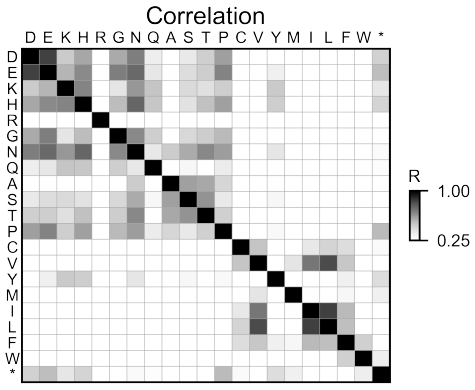
```
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

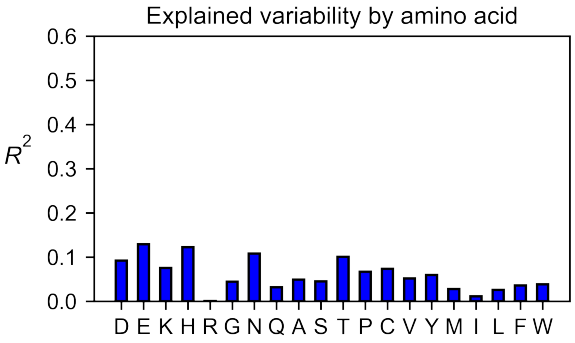


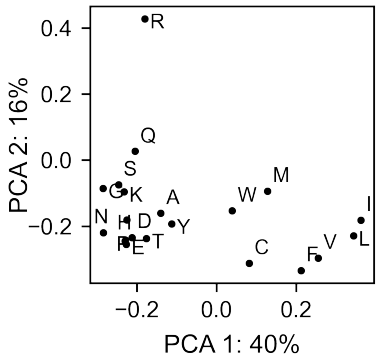





REV
KDE
Probability density
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
ΔE_x^i


Correlation
D E K H R G N Q A S T P C V Y M I L F W *
R
1.00
0.25


Explained variability by amino acid
R^2
0.6
0.5
0.4
0.3
0.2
0.1
0.0
D E K H R G N Q A S T P C V Y M I L F W


PCA 2: 16%
PCA 1: 40%
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
R
Q
S
G
K
A
N
H
D
Y
P
E
T
C
W
M
F
V
I
L

4.8.9 α-synuclein

Load data

```
#https://www.uniprot.org/uniprot/P37840#sequences
#https://doi.org/10.1038/s41589-020-0480-6

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_asynuclein'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mureved, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_asynuclein'].columns[0]

# Full sequence
sequence_asynuclein =
↪'MDVFMKGLSKAKEGVVAAAEKTKQGVAEAAGKTKEGVLYVGSKTKEGVVHGVATVAEKTK' +
↪'EQVTNVGGAVVTGVTAVAQKTVEGAGSIAAATGFVKKDQLGKNEEGAPQEGILEDMPVDP' +
↪'DNEAYEMPSEEGYQDYEPEA'

# Define secondary structure
secondary_asynuclein = [['L1'] * (1), ['α1'] * (37 - 1), ['L2'] * (44 -
↪ 37),
                        ['α2'] * (92 - 44), ['L3'] * (140 - 92)]

asynuclein_obj: Screen = Screen(
    DEMO_DATASETS['df_asynuclein'], sequence_asynuclein, aminoacids,␣
↪start_position, 0,
    secondary_asynuclein
)
```

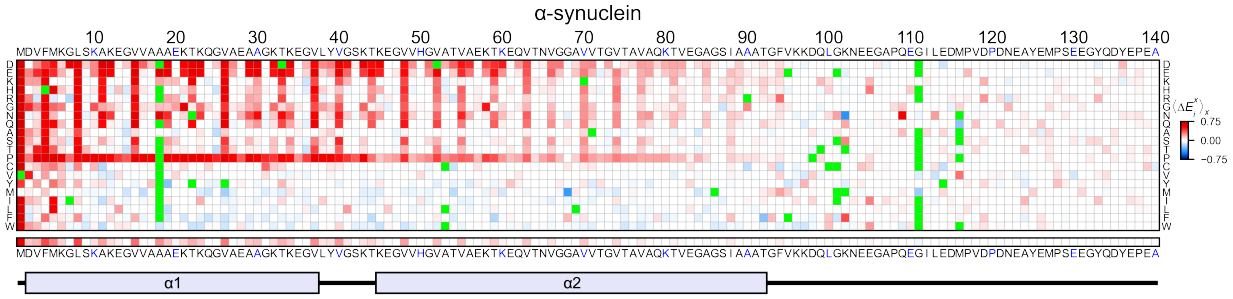
2D Plots

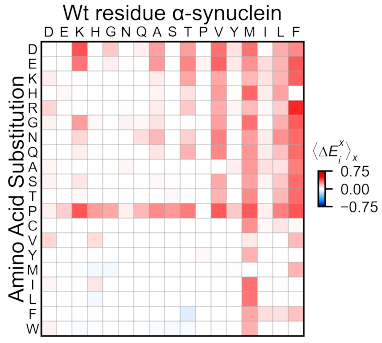
```
# Create full heatmap
asynuclein_obj.heatmap(
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids,
    title='α-synuclein',
    show_cartoon=True,
)

# Miniheatmap
asynuclein_obj.miniheatmap(
    title='Wt residue α-synuclein',
```

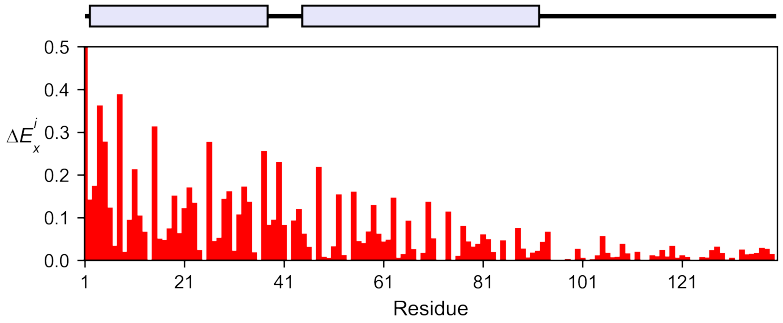
(continues on next page)

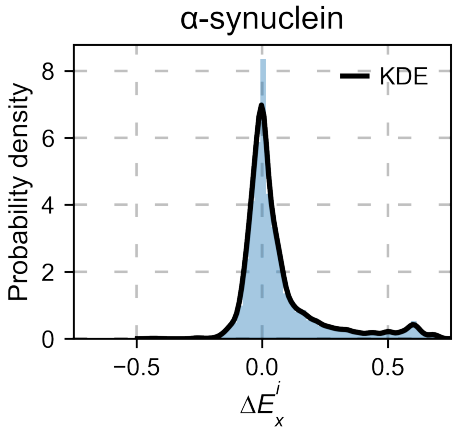
(continued from previous page)

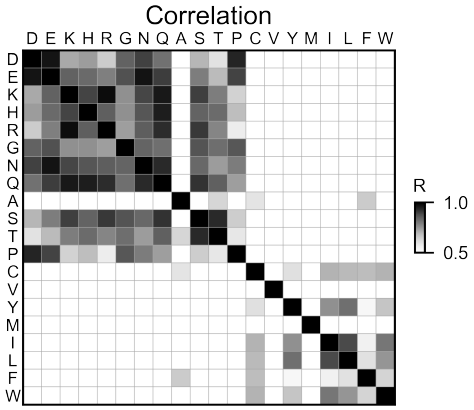
```
    colorbar_scale=(-0.75, 0.75),
    neworder_aminoacids=neworder_aminoacids,
)

# Positional mean
asynuclein_obj.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[0, 0.5],
    title='',
)

# Kernel
asynuclein_obj.kernel(
    histogram=True, title='α-synuclein', xscale=[-0.75, 0.75], output_
↪file=None
)

# Correlation between amino acids
asynuclein_obj.correlation(
    colorbar_scale=[0.5, 1],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
asynuclein_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
asynuclein_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

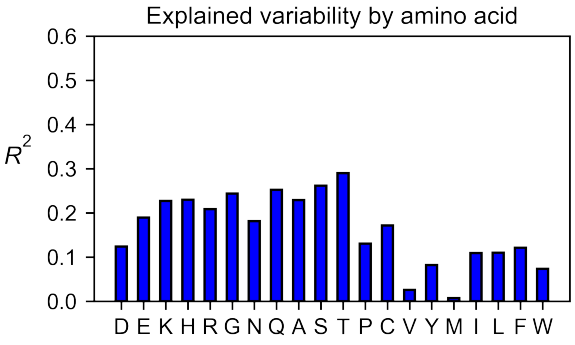

α-synuclein
MDVFMKGLSKAKEGVVAAAEKTKQGVAEAAGKTKEGVLYVGSKTKEGVVHGVATVAEKTKEQVTNVGGAVVTGVTAVAQKTVEGAGSIAAATGFVKKDQLGKNEEGAPQEGILEDMPVDPDNEAYEMPSEEGYQDYEPEA
α1
α2

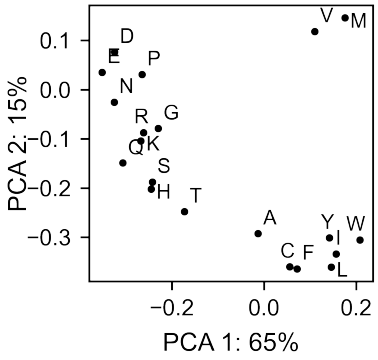

Wt residue α-synuclein
Amino Acid Substitution
0.75
0.00
−0.75


Residue


α-synuclein
KDE
Probability density







4.8.10 APH(3) II

Create object

```
#https://doi.org/10.1093/nar/gku511

aminoacids = list(DEMO_DATASETS['df_aph'].index)

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mureved, the dataset starts at residue 2
```

(continues on next page)

(continued from previous page)

```
start_position = DEMO_DATASETS['df_aph'].columns[0]

# Full sequence
sequence_aph =
 ↪'MIEQDGLHAGSPAAWVERLFGYDWAQQTIGCSDAAVFRLSAQGRPVLFVKTDLSGALNELQ' +
 ↪'DEAARLSWLATTGVPCAAVLDVVTEAGRDWLLLGEVPGQDLLSSHLAPAEKVSIMADAMRR' +
 ↪'LHTLDPATCPFDHQAKHRIERARTRMEAGLVDQDDLDEEHQGLAPAELFARLKARMPDGED' +
 ↪'LVVTHGDACLPNIMVENGRFSGFIDCGRLGVADRYQDIALATRDIAEELGGEWADRFLVLY' +
 ↪'GIAAPDSQRIAFYRLLDEFF'

# Define secondary structure
secondary_aph = [['L1'] * (11), ['α1'] * (16 - 11), ['L2'] * (22 - 16),
 ↪ ['β1'] * (26 - 22),
                  ['L3'] * (34 - 26), ['β2'] * (40 - 34), ['L4'] *␣
 ↪(46 - 40), ['β3'] *
                  (52 - 46), ['L5'] * (58 - 52), ['α2'] * (72 - 58),␣
 ↪['L6'] * (79 - 72),
                  ['β4'] * (85 - 79), ['L7'] * (89 - 85), ['β5'] *␣
 ↪(95 - 89),
                  ['L8'] * (99 - 95), ['β6'] * (101 - 99), ['L9'] *␣
 ↪(107 - 101),
                  ['α3'] * (131 - 107), ['L10'] * (135 - 131), ['α4
 ↪'] * (150 - 135),
                  ['L11'] * (158 - 150), ['α5'] * (163 - 158), ['L12
 ↪'] * (165 - 163),
                  ['α6'] * (177 - 165), ['L13'] * (183 - 177), ['β7
 ↪'] * (187 - 183),
                  ['L14'] * (191 - 187), ['α7'] * (194 - 191), ['L15
 ↪'] * (1),
                  ['β8'] * (199 - 195), ['L16'] * (201 - 199), ['β9
 ↪'] * (206 - 201),
                  ['L17'] * (212 - 206), ['β10'] * (216 - 212), ['α8
 ↪'] * (245 - 216),
                  ['L18'] * (4), ['α9'] * (264 - 249)]

aph_obj: Screen = Screen(
    np.log10(DEMO_DATASETS['df_aph']), sequence_aph, aminoacids, start_
 ↪position, 0,
    secondary_aph
)
```

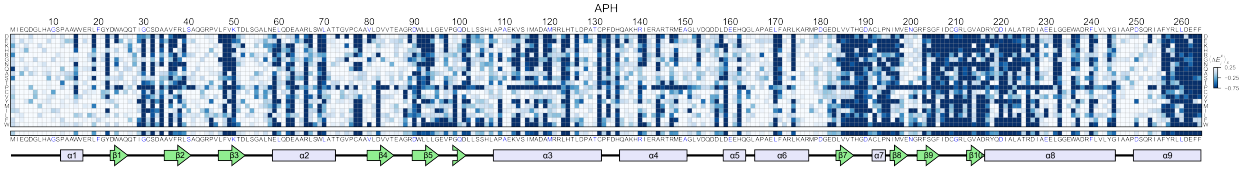
2D Plots

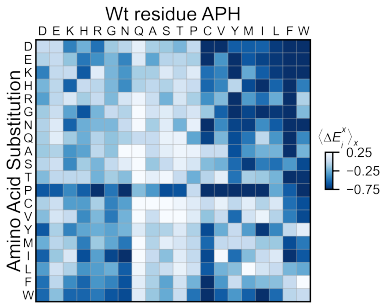
```
colormap = copy.copy((plt.cm.get_cmap('Blues_r')))

# Create full heatmap
aph_obj.heatmap(
    colorbar_scale=(-0.75, 0.25),
    neworder_aminoacids=neworder_aminoacids,
    title='APH',
    show_cartoon=True,
    colormap=colormap,
)

# Miniheatmap
aph_obj.miniheatmap(
    title='Wt residue APH',
    neworder_aminoacids=neworder_aminoacids,
    colormap=colormap,
    colorbar_scale=(-0.75, 0.25),
)

# Positional mean
aph_obj.enrichment_bar(
    figsize=[10, 2.5],
    mode='mean',
    show_cartoon=True,
    yscale=[-1.5, 0.5],
    title='',
)

# Kernel
aph_obj.kernel(histogram=True, title='APH', xscale=[-2, 2], output_
↪file=None)

# Graph bar of the mean of each secondary motif
aph_obj.secondary_mean(
    yscale=[-1, 0],
    figsize=[5, 2],
    title='Mean of secondary motifs',
)

# Correlation between amino acids
aph_obj.correlation(
    colorbar_scale=[0.25, 0.75],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
```

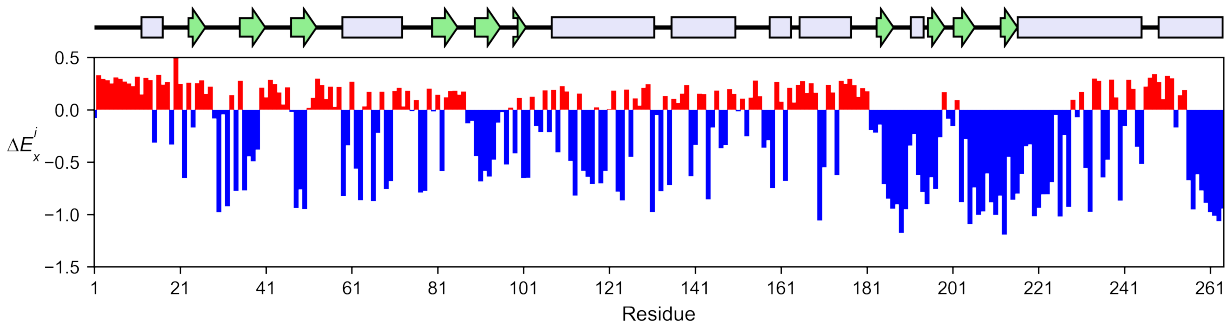
(continues on next page)

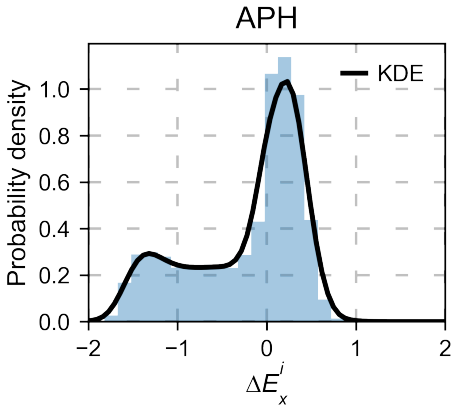
(continued from previous page)

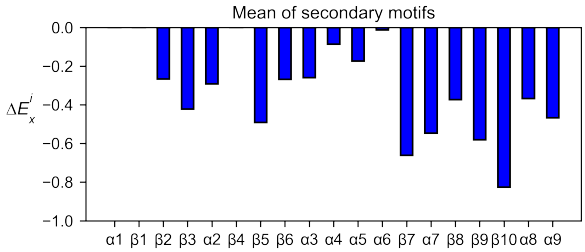
```
)

# Explained variability by amino acid
aph_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
)

# PCA by amino acid substitution
aph_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)

# PCA by secondary structure motif
aph_obj.pca(
    title='',
    mode='secondary',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
)
```

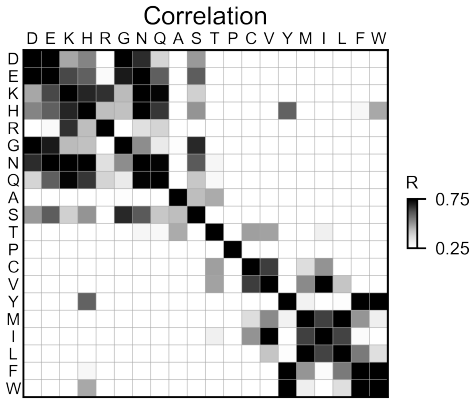


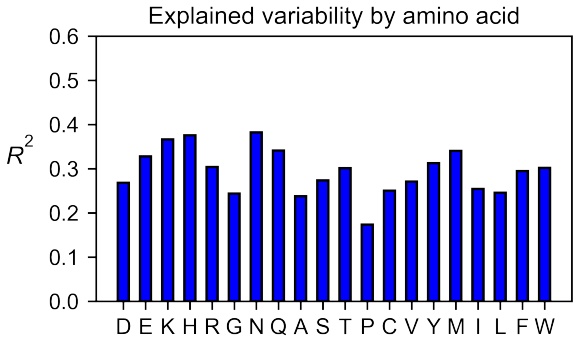


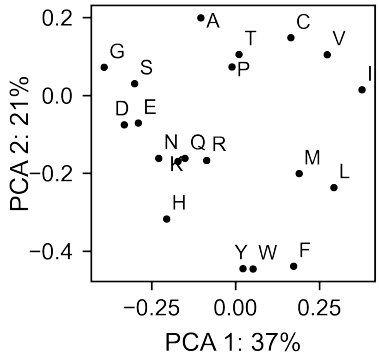

0.5
0.0
−0.5
−1.0
−1.5
ΔE_x^i
1
21
41
61
81
101
121
141
161
181
201
221
241
261
Residue

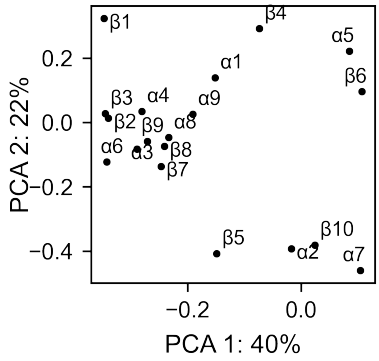

APH
KDE
Probability density
1.0
0.8
0.6
0.4
0.2
0.0
−2
−1
0
1
2
ΔE_x^i


Mean of secondary motifs
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
ΔE_x^i
α1 β1 β2 β3 α2 β4 β5 β6 α3 α4 α5 α6 β7 α7 β8 β9 β10 α8 α9


Correlation
D E K H R G N Q A S T P C V Y M I L F W
R
0.75
0.25







3D plots

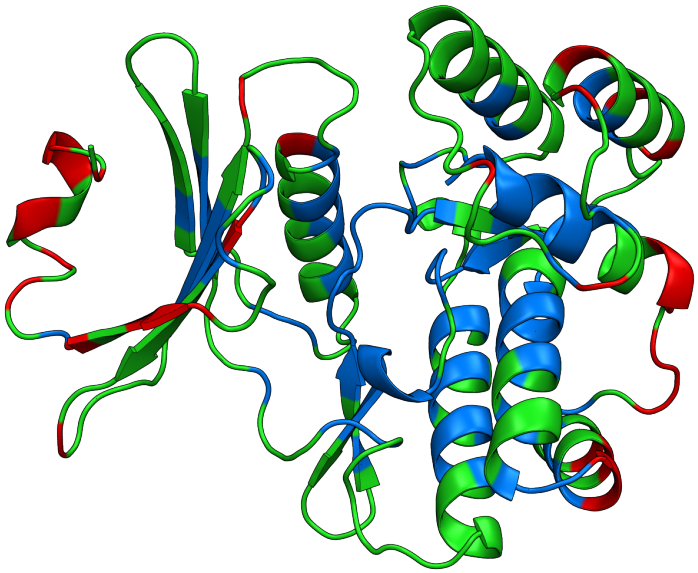
```
colormap = copy.copy((plt.cm.get_cmap('Blues_r')))

# Plot 3-D plot
aph_obj.plotly_scatter_3d(
    mode='mean',
    pdb_path=PDB_1ND4,
    title='Scatter 3D aph',
    squared=False,
    position_correction=0,
    x_label='x',
```

(continues on next page)

(continued from previous page)

```
    y_label='y',
    z_label='z',
    colormap = colormap,
    colorbar_scale = (-.75, 0.25),
)

# Plot 3-D of distance to center of protein, SASA and B-factor
aph_obj.plotly_scatter_3d_pdbprop(
    plot=['Distance', 'SASA', 'log B-factor'],
    position_correction=0,
    pdb_path=PDB_1ND4,
    title='Scatter 3D - PDB properties',
    colorbar_scale = (-.75, 0.25),
    colormap = colormap,
)
```

```
# Start pymol and color residues. Cut offs are set with gof and lof␣
↪parameters.
aph_obj.pymol(
    pdb=PDB_1ND4,
    mode='mean',
    gof=0.25,
    lof=-0.5,
    position_correction=0
)
```



4.8.11 b11l5f

Create object

```
#https://doi.org/10.5281/zenodo.1216229

# Order of amino acid substitutions in the hras_enrichment dataset
aminoacids = list(DEMO_DATASETS['df_b11l5f'].index)
neworder_aminoacids: List[str]  = list('DEKHRGNQASTPVYMILFW')

# Sequence
sequence_b11l5f = 'CRAASLLPGTWQVTMTNEDGQTSQGQMHFQPRSPYTLDVKAQGTISDGRPI
↪' + 'SGKGKVTCKTPDTMDVDITYPSLGNMKVQGQVTLDSPTQFKFDVTTSDGSKVTGTLQRQE'

# First residue of the hras_enrichment dataset. Because 1-Met was not␣
↪mureved, the dataset starts at residue 2
start_position = DEMO_DATASETS['df_b11l5f'].columns[0]

b11l5f_obj: Screen = Screen(DEMO_DATASETS['df_b11l5f'], sequence_
↪b11l5f, aminoacids, start_position, 0)
```

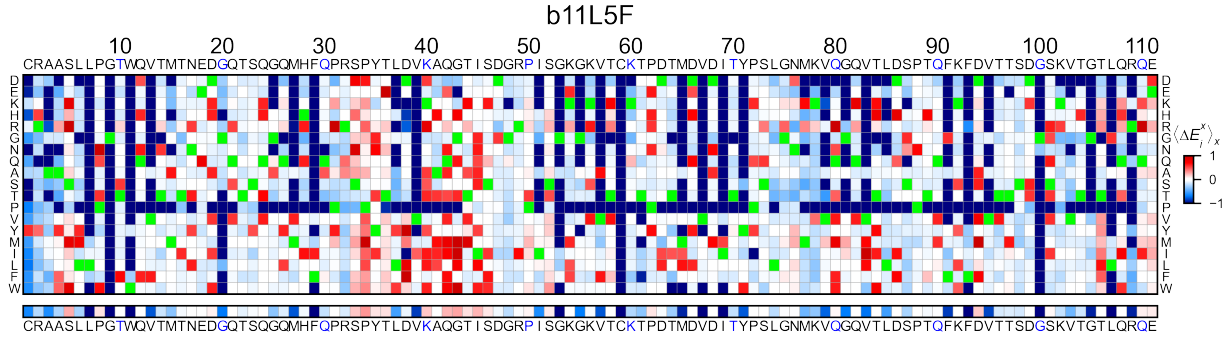
2D Plots

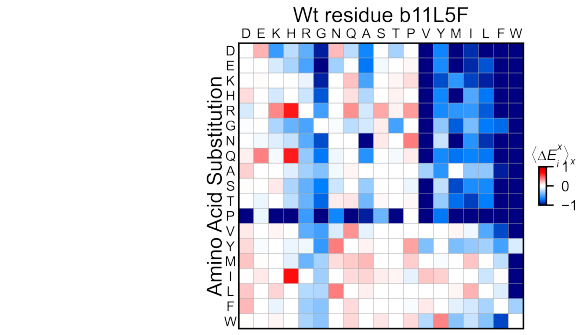
```
colormap = copy.copy((plt.cm.get_cmap('bwr')))

# Create full heatmap
b11l5f_obj.heatmap(
    neworder_aminoacids=neworder_aminoacids, title='b11l5f', output_
↪file=None
)

# Miniheatmap
b11l5f_obj.miniheatmap(
    title='Wt residue b11l5f',
    neworder_aminoacids=neworder_aminoacids,
)

# Positional mean
b11l5f_obj.enrichment_bar(
    figsize=[6, 2.5],
    mode='mean',
    yscale=[-1.5, 0.5],
    title='',
)
```

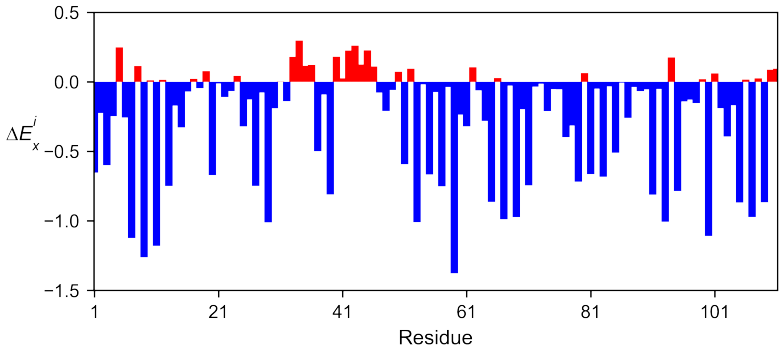
(continues on next page)

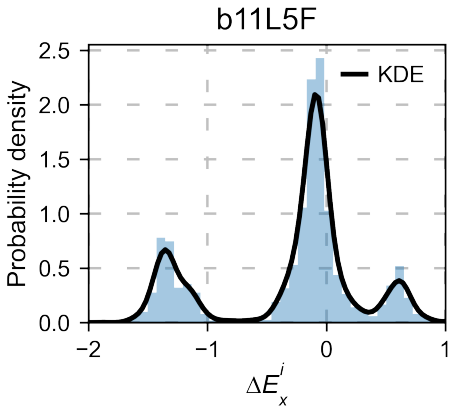
(continued from previous page)

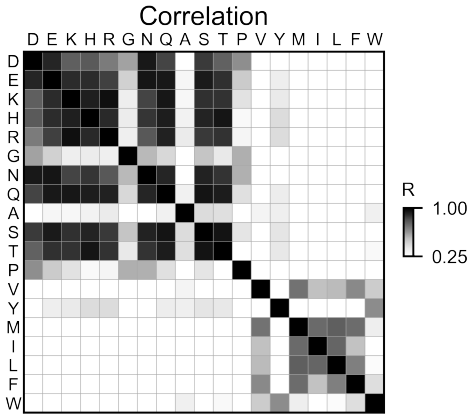
```
# Kernel
b1115f_obj.kernel(
    histogram=True, title='b1115f', xscale=[-2, 1], output_file=None
)

# Correlation between amino acids
b1115f_obj.correlation(
    colorbar_scale=[0.25, 1],
    title='Correlation',
    neworder_aminoacids=neworder_aminoacids,
)

# Explained variability by amino acid
b1115f_obj.individual_correlation(
    yscale=[0, 0.6],
    title='Explained variability by amino acid',
    neworder_aminoacids=neworder_aminoacids,
)
# PCA by amino acid substitution
b1115f_obj.pca(
    title='',
    dimensions=[0, 1],
    figsize=(2, 2),
    adjustlabels=True,
    neworder_aminoacids=neworder_aminoacids,
)
```

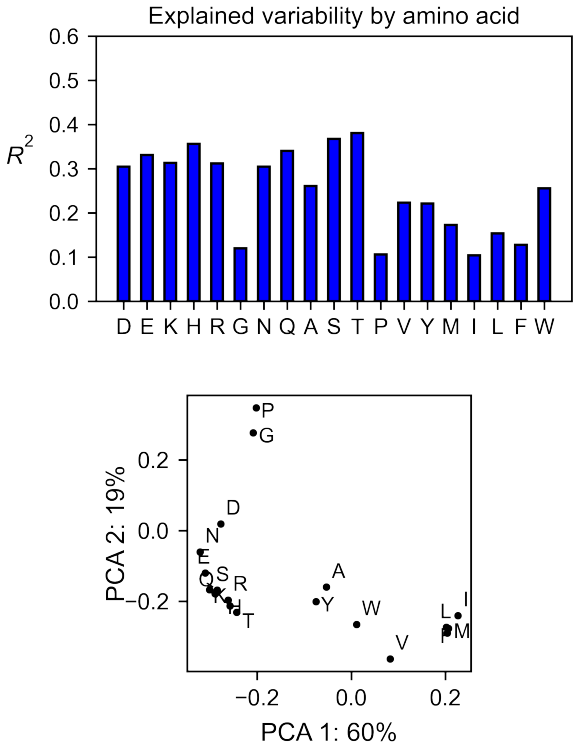



Wt residue b11L5F
D E K H R G N Q A S T P V Y M I L F W
Amino Acid Substitution
1
0
−1


0.5
0.0
−0.5
−1.0
−1.5
1
21
41
61
81
101
Residue


b11L5F
KDE
Probability density
2.5
2.0
1.5
1.0
0.5
0.0
−2
−1
0
1


Correlation
D E K H R G N Q A S T P V Y M I L F W
R
1.00
0.25



4.8.12 References

The raw data was extracted from published material. Here are the sources: beta lactamase[6] , sumo1 and ube2i[7] , mapk1[3] , tat and rev[2] , alpha-synuclein[5] , aph(3)II[4] , b1115f[1]).

[6] Stiffler, M. A., Hekstra, D. R., & Ranganathan, R. (2015). Evolvability as a function of purifying selection in TEM-1 β-lactamase. Cell, 160(5), 882–892. doi:10.1016/j.cell.2015.01.035

[7] Weile, J., Sun, S., Cote, A. G., Knapp, J., Verby, M., Mellor, J. C., … Roth, F. P. (2017). A framework for exhaustively mapping functional missense variants. Molecular Systems Biology, 13(12), 957. doi:10.15252/msb.20177908

[3] Livesey, B. J., & Marsh, J. A. (2020). Using deep mutational scanning to benchmark variant effect predictors and identify disease mutations. Molecular Systems Biology, 16(7), e9380. doi:10.15252/msb.20199380

[2] Fernandes, J. D., Faust, T. B., Strauli, N. B., Smith, C., Crosby, D. C., Nakamura, R. L., … Frankel, A. D. (2016). Functional segregation of overlapping genes in HIV. Cell, 167(7), 1762–1773.e12. doi:10.1016/j.cell.2016.11.031

[5] Newberry, R. W., Leong, J. T., Chow, E. D., Kampmann, M., & DeGrado, W. F. (2020). Deep mutational scanning reveals the structural basis for α-synuclein activity. Nature Chemical Biology, 16(6), 653–659. doi:10.1038/s41589-020-0480-6

[4] Melnikov, A., Rogov, P., Wang, L., Gnirke, A., & Mikkelsen, T. S. (2014). Comprehensive mutational scanning of a kinase in vivo reveals substrate-dependent fitness landscapes. Nucleic Acids Research, 42(14), e112. doi:10.1093/nar/gku511

[1] Dou, J., Vorobieva, A., Sheffler, W., Doyle, L., Park, H., Bick, M., … Baker, D. (2018). De Novo Design Of A Fluorescence-Activating B-Barrel. Zenodo. doi:10.5281/zenodo.1216229

CHAPTER 5

About Us

Get to know more about the *Frank Hidalgo*, *Sage Templeton*, *Joanne Wang*, and *Che Olavarria Gallegos*.

5.1 About the authors

5.1.1 Frank Hidalgo

Frank is currently a 5th year Chemical Biology PhD student at UC Berkeley. He is a self-directed and data-driven scientist interested in using cutting-edge computational techniques to solve problems at the interface between health and technology. After he graduates from Cal, he is looking forward to joining a multidisciplinary team where he can put the skills he has learnt in data science and biology at work.

Frank is also part of Beyond Academia (BA), a non-profit organization. BA's mission is to empower graduate students and postdocs to expand their career options beyond the traditional academic track. Check out the events organized by BA on their website.

Feel free to contact Frank on LinkedIn .

5.1.2 Sage Templeton

Sage is currently an undergraduate senior majoring in Molecular and Cell Biology: Biochemistry and Molecular Biology and Public Health at UC Berkeley. She is interested in the analysis of living

systems at all levels of organization: from genetic changes in minute proteins such as KRAS to the effects of our climate crisis on population health. Sage is an undergraduate researcher in the Kuriyan Lab studying protein stability in HRAS under her mentor Frank Hidalgo. After graduation from Cal in 2021, Sage plans to return to her home state of New Mexico to work in community health policy before attending medical school.

5.1.3 Che Olavarria Gallegos

Che is currently an undergraduate senior majoring in chemistry at Carnegie Mellon. He is interested in the using software engineering and computational chemistry to aid in synthesis. Che plans to attend graduate school after graduating.

5.1.4 Joanne Wang

Joanne is currently a second year undergraduate student at UC Berkeley intending to major in Molecular and Cell Biology with an emphasis in Neurobiology. She plans to work in healthcare or biotech in the future.

Last year as a freshman in UC Berkeley's Kuriyan Lab, Joanne conducted research alongside her mentor Frank Hidalgo to assess the protein stability of various single-point H-RAS mutants. This semester she is working remotely from San Diego and has transitioned her focus onto a computational project. Joanne has learned and been using python to improve the ways mutagenesis data can be analyzed which is applicable to her previous work with H-RAS mutants.

Index

Symbols

C

D

E

G

H

I

K

L

M

P

R

S