ProPythia - User Guide

Ana Marta Sequeira

March 2021

1 Introduction

This document intends to provide an overview of both the package structure, functions with a small theoretical approach. If you find mistakes or have improvement suggestions please contact.

2 Motivation and target audience

To facilitate and foster the extraction of relevant knowledge from protein sequences, we developed ProPythia, a generic, comprehensive and versatile semi-automated platform to ease the classification of peptides/proteins. ProPythia provides functions to calculate a variety of physicochemical properties and other protein representations, implementing around 40 types of descriptors. It enables to perform dataset preprocessing, managing and performing changes in sequences, clustering, manifold learning, feature selection and dimensionality reduction with a variety of diagrams to facilitate user interpretability. Also, it allows the training and optimization of traditional ML models covering the main classes of algorithms, and their use to make predictions over unseen data and respective feature analysis.

Besides, ProPythia makes available a DL pipeline with baseline models, including feedforward Deep Neural Networks (DNNs), Recurrent Neural Networks such as Long-Short Term Memory (LSTM) networks, Convolution Neural Networks (CNN), embedding layers, and possible combinations of these architectures in the different layers of hybrid models. Also, it allows the user to create her/his own model, offering the possibility to not only develop the models, but to optimize and evaluate them, construct several plots and make predictions.

Regarding classification tasks, ProPythia can be used for binary and multi-class contexts. In addition, it has built-in reporting functions, that can also export all results directly to files. As it is built in a modular way, the user retains the power to manipulate and use other functions outside of the package, having control over the different steps and being able to adapt/extend the code to fit specific needs. The package is directed to handle protein related problems, but its modular construction allows users to use it in other problems.

- It offers the option to change the sequences and obtain subsequences (such as N and C terminals or scanning windows)
- It can extract/calculate a high number and type of descriptors for protein sequences;
- It is designed to conduct all main steps to construct a predictor, facilitating the machine learning process in all of its stages (which is not common in the available packages);
- It can be used for ML and DL. It has a variety of possible Shallow algorithms to implement and a complete DL framework where the user can design is own model or used pre established ones.
- It is, compared to others, more user friendly with more variety of plots and schemes to help cluster, features and machine learning analyses;

- It is built in modular ways, which is also a major advantage, as the user can, easily integrate other features or use other methods of different tools to complete their work

It is intend to be used by researchers from different bio fields, that not having deep knowledge of Machine learning, want to explore Machine Learning in a simple form with protein sequences.

3 Code and package structure

All the code developed was written in python 3.6 using the *Anaconda Data science* platform and *Pycharm professional 2019.1* as IDE. The most important libraries used were *Scikit-learn*, *NumPy*, *SciPy* and *pandas*. Additionally, the packages of *modlAMP* [11], *PyDPI* [2], *pfeature* [13] and *Biopython* [5] were used. The ML/ DL algorithms were implemented using mainly *Scikit-learn* [14], *Tensorflow* [1] and *Keras* [**chollet2015**] libraries.

The package was built in a modular way, allowing users to have control over the different steps and to adapt/extend the code to fit their specific needs. In order to use the modules, it is necessary to create a class object and call the desired methods from each of the modules. The user can set specific values for the majority of the parameters, but default values are established.

The package is composed by the following modules:

1. **sequence module**: to read and/or change sequences or generate subsequences;
2. **descriptors module**: to compute different types of protein descriptors;
3. **preprocess module**: to do a preprocessing of the feature vectors and 'clean' the dataset;
4. **dimensionality reduction module**: to perform dimensionality reduction using Principal Component Analysis (PCA), batch sparse PCA and truncated singular value decomposition (SVD);
5. **manifold module**: to perform non-linear dimensionality reduction suitable for data visualization. t-SNE and UMAP avaialable;
6. **feature_selection module**: to discover and select the most valuable features using univriate or model base approaches;
7. **clustering module**: to perform and plot the results of a clustering analysis;
8. **machine_learning module**: to implement supervised machine learning algorithms;
9. **deep_learning module**: to implement supervised DL algorithms for classification, encompassing models using different types of input representation.

Additionally, the package contains a test module to validate if all the functions are working properly and example files of a possible implementation of a pipeline using the package (Jupyter notebook). A schematic view of the package can be seen in Figure 1.

Below, is presented a brief description of all the models. For more information please see Appendix.

3.1 sequence module

The read sequence module contains the `ReadSequence` class that is used to read or change sequences and obtain subsequences. The class allows to:

- Read sequences from string or from an Uniprot ID (it is also possible to retrieve sequences from text with UniProt IDs) retrieving sequence objects used to calculate descriptors (following module);

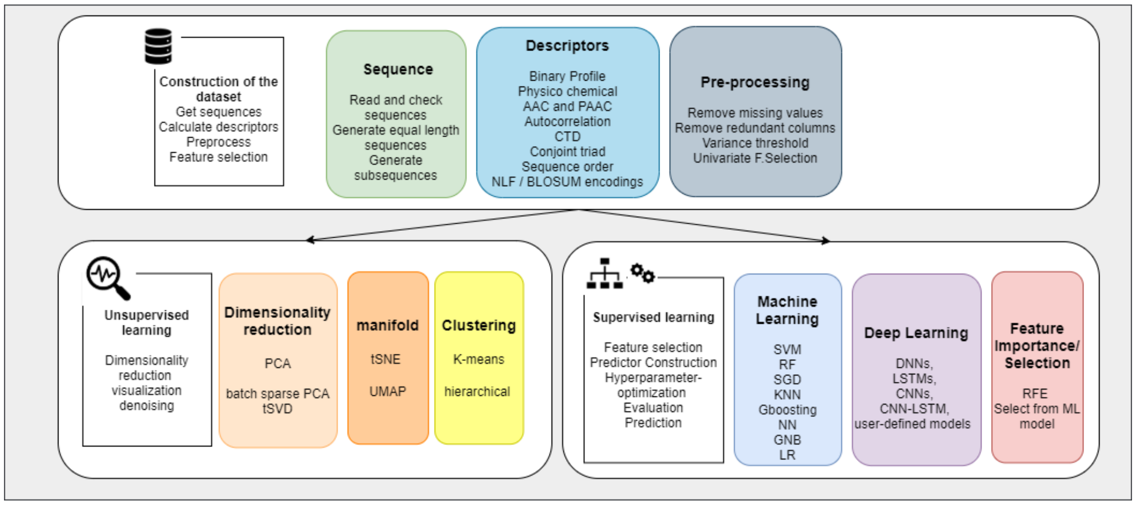

Figure 1: Schematic representation of the modules in the built package

- Check if it is a valid *aminoacid (aa)* sequence;
- Obtain sequences with a given size from a list of sequences, adding or cutting from both the N and C terminals. This may be specially relevant to calculate features that are length sequence dependent;
- From one sequence, generate a list of sub-sequences based on a sliding window approach, from specific aa, from the terminals or dividing the sequence in parts. Beginning with only one sequence it will generate a list of sub sequences. A sliding window approach can be particularly helpful in screening sites problems, the division by terminals and from specific AA are most used in biological approaches.

A summary table is show below (Table 1).

Table 1: Summary table of the methods available in the class `ReadSequence`

Class ReadSequence	**Aim**	**Method**
Read sequences	Read a protein sequence	*read_protein_sequence()*
	Downloading a protein sequence by uniprot id	*get_protein_sequence_from_id()*
	retrieve sequences from a txt with uniprot ID	*get_protein_sequence_from_txt()*
Check protein	Check if sequence is a valid aminoacid sequence	*checkprotein()*
Equal length	Cut or add aa to obtain sequences with equal length	*get_sized_seq()*
Generate sub seqs (1 $\rightarrow$ n sequences)	Sliding window of the protein given. It will generate a list of n sequences with length equal to the value of window and spaced by a gap value	*get_sub_seq_sliding_window()*
	Get all 2*window+1 sub-sequences whose center is ToAA in a protein	*get_sub_seq_to_aa()*
	Split the original seq in n number of subsequences	*get_sub_seq_split()*
	Divide the sequence in the N terminal and C terminal with sizes defined by the user	*get_sub_seq_terminals()*

3.2 Descriptors module

The descriptors module aims to compute different types of protein descriptors and is carried out with the class named `Descriptor`. The descriptors functions are retrieved from the packages *Biopython*, *modlAMP*, *pfeature* and *PyDPI*. The class takes as input a protein sequence (from the previous model) that is used to calculate a variety of protein descriptors.

The descriptors are divided into categories and it is possible to calculate the descriptors with the individual functions, all the functions in same category or calculate all the descriptors available by calling the *get_all* function. If the user desires to choose the features, the function *adaptable* receives as input a list containing the numbers of the desired descriptors to be calculated.

Please bee aware that the aminoacids supported to the calculus of features are:
"A", "R", "N", "D", "C", "E", "Q", "G", "H", "I", "L", "K", "M", "F", "P", "S", "T", "W", "Y", "V".
Please be aware that some characters that may appear are not supported, including the 'X' (used to fill sequences when the aminoacid is unknown), 'B' (which can be representative of an asparagine or aspartic acid), 'Z' (which can be representative of glutamine or glutamic acid), 'U' (selenocysteine, an aa structurally close to cysteine) and 'O' (pyrrolysine, an aa structurally close to lysine).

The `Descriptor` class allows the calculation of:

- Binary profiles for both sequence and for 25 physicochemical properties of each residue of the sequence;
- Physicochemical descriptors. It has 16 features available that include length, charge, charge density, formula (number of C,H,N,O and S in the sequence), number of 4 types of molecular bonds (total, single,double and hydrogen bonds), the molecular weight, *grand average of hydropathy (GRAVY)*, aromacity, isoelectric point, instability index, values for secondary

structures (a-helix, turns and b-sheets), molar extinction coefficient, flexibility, aliphatic index, boman index and hydrophobic ratio;

- Aminoacid composition functions. It retrieves the aminoacid composition, the Dipeptide composition, and the tripeptide composition;
- Pseudo aminoacid composition and the amphiphilic aminoacid composition;
- Autocorrelation descriptors including Normalized Moreau-Broto autocorrelation, Moran Autocorrelation and Geary autocorrelation values;
- Composition, Transition and Distribution descriptors;
- Conjoint Triad descriptors;
- Sequence order descriptors. It calculates sequence order coupling numbers and quasi sequence order values;
- Base class peptide descriptors. It allows to calculate the sequence moment, the global averaging descriptor, hydrophocity moments, arcs, autocorrelation and cross correlation of amino acid values for a given descriptor value.
- aminoacid encodings using BLOSUM (62 and 50) substitution matrice or NLF scoring matrice.

Overall, 40 descriptor and 8 agglomerative functions are available. A summary table can be seen below (Table 2).

Table 2: Summary table of the functions available in the module Descriptor. K means the length of the sequence

Class Descriptor	**Nº**	**Aim**	**Method**	**NºDescriptors**
Binary Profile	1	Binary profile of aminoacid composition	*getbin_aa()*	20*k
	2	Binary profile of residues for 25 phychem feature	*getbin_resi_prop()*	Max 25*k
Physico chemical	3	Length of sequence	*get_length()*	1
	4	Charge of sequence	*get_charge()*	1
	5	Charge density of sequence	*get_charge_density()*	1
	6	Number of C,H,N,O and S of the aa of sequence	*get_formula()*	5
	7	Sum of the bond composition for each type of bond	*get_bond()*	4
	8	Molecular weight	*get_mw()*	1
	9	Gravy from sequence	*get_gravy()*	1
	10	Aromacity	*get_aromacity()*	1
	11	Isolectric Point	*get_isoelectric_point()*	1
	12	Instability index from sequence	*get_instability_index()*	1
	13	Fraction of aa which tend to be in helix, turn or sheet	*get_sec_struct()*	3
	14	Molar extinction coefficient	*get_molar_extinction_coefficient()*	2

Class Descriptor	Nº	Aim	Method	NºDescriptors
	15	Flexibility according to Vihinen, 1994	*get_flexibility()*	-
	16	Aliphatic index of sequence	*get_aliphatic_index()*	1
	17	Boman index of sequence	*get_boman_index()*	1
	18	Hydrophobic ratio of sequence	*get_hydrophobic_ratio()*	1
	19	All 15 geral descriptors	*get_all_physicochemical()*	-
Aminoacid Composition	20	Aminoacid compositon	*get_aa_comp()*	20
	21	Dipeptide composition	*get_dp_comp()*	400
	22	Tripeptide composition	*get_tp_comp()*	8000
	23	All descriptors from Aminoacid Composition	*get_all_aac()*	8420
Pseudo aminoacid Composition	24	Type I Pseudo aminoacid composition	*get_paac()*	Min 30, depends lambda
	25	Type I Pseudo aminoacid composition for a given property	*get_paac_p()*	Min 30, depends lambda
	26	Type II Pseudo aminoacid composition - Amphiphilic	*get_apaac()*	Min 30, depends lambda
	27	Calculate PAAC and APAAC	*get_all_apaac()*	Min 60
Auto correlation	28	Normalized Moreau-Broto autocorrelation	*get_moreau_broto_auto()*	240
	29	Moran autocorrelation	*get_moran_auto()*	240
	30	Geary autocorrelation	*get_geary_auto()*	240
	31	Calculate all descriptors from Autocorrelation	*get_all_correlation()*	720
CTD	32	Composition Transition Distribution	*get_ctd()*	147
Conjoint Triad	33	Conjoint Triad	*get_conj_t()*	343
Sequence Order	34	Sequence order coupling numbers	*get_socn()*	90 (default)
	35	Sequence order coupling numbers	*get_socn_p()*	90 (default)
	36	Quasi sequence order	*get_qso()*	100 (default)
	37	Quasi sequence order	*get_qso_p()*	100 (default)
	38	Calculate all values for sequence order descriptors	*get_all_sequenceorder()*	190 (default)
Base Class Peptide	39	Moment of sequence	*calculate_moment()*	1
	40	Global / window averaging descriptor	*calculate_global()*	1
	41	Hydrophobicity or hydrophobic moment profiles	*calculate_profile()*	2
	42	Calculates arcs	*calculate_arc()*	5

Class Descriptor	Nº	Aim	Method	NºDescriptors
	43	Autocorrelation of aa values for a given descriptor scale	*calculate_autocorr()*	-
	44	Cross correlation of aa values for a given descriptor scale	*calculate_crosscorr()*	-
	45	All functions from Base class	*get_all_base_class()*	-
Other encodings	46	NLF encoding	*get_nlf_encode()*	1
	47	Blosum encoding	*get_blosum()*	1
All	48	Calculate all possible descriptors (except 1,2,23, 47,48)	*get_all()*	-
Adaptable	49	Choose which functions to calculate	*adaptable()* (list_of_functions=[])	-

3.3 preprocess module

The preprocessing module allows the transformation of the feature vectors into representations suitable for downstream estimators. By 'cleaning' the dataset from redundant features, it removes a large number of non relevant columns from the dataset. It is callable by the class `Preprocess` and allows for the:

- Elimination of columns with only zero values;
- Elimination of repeated columns;
- Elimination of low variance columns (by default zero variance, but the user can adapt, this function has an imbued scaler).

It is possible to call a single function *preprocess* to run all the functions above. The functions names and aims can be seen in Table 3.

Table 3: Summary table of the methods available in the class `Preprocess`

Class Preprocess	Aim	Method
Check data	Check nans	*missing_data()*
Remove low relevance columns	Remove columns that have all values as zero	*remove_columns_all_zeros()*
	Remove duplicated columns	*remove_duplicate_columns()*
	Remove all features whose variance does not meet some threshold	*remove_low_variance()*
	Remove columns that have all values as zero, duplicated and low variance columns	*preprocess()*

Unsupervised algorithms can be useful for data visualization and understanding and dimensionality reduction. ProPythia has 3 modules dedicated to this type of algorithms.

3.4 linear_dim_reduction module

In *Machine learning (ML)*, feature reduction techniques allow to summarize the essential characteristics of a high dimensional data representation reducing the number of features [12]. The module *linear_dim_reduction* has the class `FeatureDecomposition` which allows to perform dimensionality reduction using Principal Component Analysis (PCA), batch sparse PCA and truncated singular value decomposition (SVD).

Principal component analysis (PCA) is a statistical procedure that orthogonally transforms the original coordinate system of a (numerical) data set into a new set of coordinates, the principal components, that capture well the variance of original features, i.e, decompose a dataset in a set of successive orthogonal components that explain a maximum amount of the variance.

TruncatedSVD is similar to PCA but does not center the data before computing the singular value decomposition, working well with sparse matrices. Mini-batch sparse PCA extracts sparse components that best reconstruct the data, iterating over small chunks of features. Sparse components allows for better interpretable representation, emphasizing which of the original features contribute to the differences between samples [14].

For better analysis and perception, this module enables the production of plots. This module has a report function, that if activated when initiated the class will store all the results in txt file.

The main purposes of this class are to:

- Derive the PCA of the data. The function receives a dataset, and applies the *Scikit-learn* PCA algorithm retrieving the fitted and the transformed dataset [6];
- Perform sparse PCA analysis using batches;
- Perform truncated SVM. This transformer performs linear dimensionality reduction by means of truncated singular value decomposition(SVD).Contrary to PCA, this estimator does not center the data before computing the singular value decomposition. This means it can work with sparse matrices efficiently.applies the algorithm to a dataset and retrieves the fitted and the transformed dataset;
- Access the contribution of each feature for the component;
- Access the variance ratio of each component;
- Produce a bar plot with percentage of explained variance ratio by components;
- Produce a bar plot to cumulative explain variance per number of components. useful to understand how many components are needed to describe the data;
- Produce a scatter plot 2D or 3D that represents the different classes based on 2 or 3 *Principal component (PC)*s (by default the first two/three) . As the PCA or SVD components are orthogonal to each other (not correlated), this usually allows to distinguish classes in a clear way.

It is worth noting that these are unsupervised method, meaning that PCA components are calculated only from features and no information from classes are considered. Besides this, each component does not represent a feature but a mixture of the original features.

A summary table of the functions that can be implemented with this class is given below (Table 4).

Table 4: Summary table of the methods available in the class `linear_dim_reduction`

Class FeatureDecomposition	Aim	Method
Algorithms	Perform the PCA analysis	*run_pca()*
	Perform Batch Sparse PCA	*run_batch_sparse_pca()*
	Perform truncated SVD	*run_truncated_svd()*
Analysis	Measure the variance ratio of the principal components	*variance_ratio_components()*
	Retrieve a dataframe containing the contribution of each feature (rows) for component	*contribution_of_features_to_components()*
Design Graphics	Derive a bar plot representing the percentage of explained variance ratio by PCA	*pca_bar_plot()*
	Plot cumulative explain variance per number of components	*pca_cumulative_explain_ratio()*
	Scatter plot of the labels based on 2 components (by default the first ones)	*pca_scatter_plot()*
	Scatter plot of the labels based on 3 components (by default the first ones)	*pca_scatter_plot3d()*

3.5 manifold module

The *manifold module* implements approaches to non-linear dimensionality reduction suitable for data visualization.

t-SNE converts affinities of data points to probabilities, is particularly sensitive to local structure being, however, computationally expensive and limited to two or three dimensional embeddings [14].

UMAP (Uniform Manifold Approximation and Projection) is a recent manifold learning technique for dimension reduction. It is scalable, being competitive with t-SNE for visualization quality, preserving more of the global structure, having better time performance and no computational restrictions on embedding dimension [**McInnes2018**]. For a better analysis and interpretation of the results, scatter plot is available.

The main purposes of this class are to:

- Perform t-SNE. t-SNE is tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results;
- Perform UMAP analysis
- Scatter plot for 2 dimensions for both t-SNE and UMAP analysis.

A summary table of the functions that can be implemented with this class is given below (Table 5).

Table 5: Summary table of the methods available in the class `Manifold`

Class Manifold	**Aim**	**Method**
Algorithms	Perform t-SNE analysis	*run_tsne()*
	Perform UMAP analysis	*run_umap()*
Plots	Scatter plot 2D for UMAP or tSNE	*manifold_scatter_plot()*

3.6 clustering module

The class `Cluster` aims to perform and plot clustering analysis. Clustering is a type of unsupervised machine learning useful to find homogenous subgroups, such that objects in the same group (cluster) are more similar to each other than with others [18]. If the labels are already available, clustering analysis can be useful to see how the samples are grouped and if they match indeed the labels defined. Besides that, clusters can also be used as features in a supervised machine learning model.

The *clustering* module allows to perform and plot the results of a K-means, mini-batch k-means and/or hierarchical clustering analysis. K-means clusters data by separate samples in a pre-defined number of groups. Mini-batch K-Means is a variant of the K-Means that uses subsets of the input data to reduce the computation time. Hierarchical clustering builds nested clusters being the output represented as a tree/ dendogram.

This class implements methods for:

- Performing K means cluster classification;
- Performing K means cluster using mini batches;
- Evaluate clustering results on target data;
- Performing and plot hierarchical clustering using scikit-learn or scipy strategies.

The k-means algorithm clusters data by dividing observations into k groups of equal variance - clusters. It can be easily used in classification where we divide data into clusters which can be equal to or more than the number of classes. It scales well to large number of samples and is one of the most widely used clustering algorithms. The algorithm divides the samples into clusters, each described by the mean of the samples of the cluster – centroids. Initially, the algorithm chooses the initial centroids randomly; then, it will loop between assigning each sample to its nearest centroid and changing the centroids by taking the mean value of all the samples assigned to each previous centroid, repeating until the centroids do not move. The function *Kmeans* applies the Kmeans algorithm to the dataset. It is possible to edit the maximum number of iterations (300 per default) and the number of clusters (by default the number of existing labels). The method implements the k-means initialization scheme to address the primary choice of the centroids distant from each other leading to better results. When dataset are large, doingcluster with mini batches may be a good solution.

The class has functions to visualize data with *hierarchical clustering (HC)* as well. Hierarchical clustering builds nested clusters by merging or splitting them successively, with this hierarchy being represented as dendograms (trees). The root of the tree is a unique cluster with all the samples and the leaves are clusters with only one sample. The HC uses a bottom up approach where each observation starts in its own cluster and where clusters are successively merged together [8]. The algorithm stops when only one cluster remains, the root.

To build the hierarchical clusters, the *SciPy* library *cluster.hierarchy* was used. It is possible to calculate the distance between clusters using the method 'single' that minimizes the distance of all the pairs of clusters, the method 'complete' that maximizes that distance or the method 'average' that represents the average of all distances. Other methods such as 'weighted', 'centroid', 'median', 'ward' are available as well [17]. The metric used to pairwise distances between observations in n-dimensional space can be chosen as well, taking values as 'correlation', 'euclidean', 'hamming' or others [10]. The function *hierarchical_dendogram* takes as input the metric and distance parameters and returns the graphic of the hierarchical division of samples. Alternatively, the HC can be build using *scikit-learn* receiving the number of clusters, type of affinity and type of linkage. This scikit HC can be then be plotted using the plot_dendrogram with different levels of truncation. A view of the methods available in this class can be seen in the table below (Table 6).

Table 6: Summary table of the methods available in the module `Cluster`

Class Cluster	**Aim**	**Method**
Kmeans	Perform K means cluster	*run_kmeans()*
	Perform K means cluster using mini batch	*run_minibatch_kmeans()*
	Evaluate the cluster results with true labels	*clust_evaluation()*
Hierarchical	Perform and plot scipy hierarchical clustering	*hierarchical_dendogram()*
	Perform HC using scikit-learn	*run_hierarchical()*
	Plot scikit learn diagram	*plot_dendrogram()*

3.7 Feature Selection module

Feature selection is another way to reduce the dimensionality of the dataset by selecting the most valuable features to improve estimators' accuracy and/or boost their performance on very high dimensional datasets [9]. This selection allows to gain knowledge and understanding of the data, quite important in biological questions [7].

There are three major methods to select features - filter methods, wrapper methods and embedded methods [15] and therefore, this package has methods to perform univariate feature selection, *recursive feature elimination (RFE)* and selection of features based on model like a tree or *support vector machines (SVM)*. The package class `Feature_selection` is based on *Scikit-learn* functions from the *feature.selection* module.

This module has a report function, that if activated when initiated the class will store all the results in txt file.

This module allows to:

- Perform univariate feature selection. The function has a configurable strategy and the user will decide which score function to use and how the features will be selected exactly like in *Scikit-learn*. It is possible to select the k highest scoring features, specified percentage of features and features based on false positive rate, false discovery rate or family wise error. The scoring functions (return scoring values and p-values) include *chi2*, *f_classif* or *mutual_info_classif*;

- Perform recursive feature elimination with or without cross validation. The user can decide the number of features to select;
- Retrieve scores from recursive feature elimination rankings;
- Select features based on embedded techniques using user selected estimators (e.g. TreeClassifier, logistic regression and linear SVC);
- Obtain the features names and scores of importance for all techniques, retrieving a dataframe with features names and scores by order of relevance.

Univariate feature selection selects the best features (high correlation with outcome variable) based on univariate statistical tests and can be used as preprocessing step. This test is only reliable if the variables are fully independent. They are independent of any machine learning algorithm, with the features being selected solely based on their correlation with the outcome variable. It is important to take into account that the methods based on F-test estimate the degree of linear dependency between two random variables whereas, mutual information methods can capture any kind of statistical dependency, but being non parametric, they require more samples for accurate estimation. *Chi2* will only work for non negative features. If the parameters are not given the function will calculate based on the scoring function *mutual_info_classif* and selecting the best 10^{-5} percentile of features. It returns the dataset fit and transformed, the new dataset and the columns selected. The higher the score the more representative is the feature.

Wrapper methods use a subset of features to train and assess the performance of a model, being the subset of features tailored to this model. RFE uses an external supervised learning estimator that provides information about feature importance (a coef_ attribute or a features_importances attribute), such as SVM; from the initial set of features, the estimator assigns weights to each feature and eliminates the ones with smallest weights, repeating the process until the desired number of features is achieved. *The recursive_feature_elimination* function allows to recursively eliminate features less important with or without cross validation. The user can decide the number of features to select. The function retrieves the dataset fit and transformed, the original dataset with the features selected, the columns names and the features ranking. Using the function *rfe_ranking* a dataframe containing the features names and the ranking by order is obtained. Rankings of 1 corresponds to the ones selected.

It is also possible to select features based on embedded techniques. Here, the optimal set of features is built into the classifier construction as the estimator retrieves a *coef_* or a feature_importances vector. The features considered unimportant, below a threshold, are removed. The threshold can be a 'median', 'mean', floats of the strings or any numerically valuable. It is possible to do this in two ways:

- Considering linear models penalized with the L1 norm for classification, such as logistic regression and linear *Support Vector Classification (SVC)*. They retrieve sparse solutions and negative values. The data should not be very noisy and the features should be independent. The most important features have the highest scores and features uncorrelated with the output variable should have coefficients close to zero. If the correlation coefficient is negative, it provides statistical evidence of a negative relationship between the variables. The increase in the first variable will cause the decrease in the second variable.
- Considering tree based estimators that compute features importance and can be used to discard irrelevant features.

A view of the methods available in this class can be seen in the table below (Table 7).

Table 7: Summary table of the methods available in the class `FeatureSelection`

Class FeatureSelection	Aim	Method
Feature selection	Univariate feature selector	*run_univariate()*
	Recursive feature elimination	*run_recursive_feature_elimination()*
	Select from model	*run_from_model_feature _elimination()*
Features scores	Retrieves a dataframe with features names and scores of importance	*scores_ranking()*
Others	Retrieves the feature selected dataset	*get_transformed_dataset()*

Supervised algorithms map input data to known targets. This section describes the available algorithms for supervised learning, including approaches from traditional ML, recent DL algorithms and methods for feature selection.

3.8 shallow_ml module

In order to implement supervised machine learning algorithms, the class `ShallowML` was created. The machine learning module intends to facilitate the application of ML models to the classification of peptides.Similarly to the rest of the package, the user can tune all the parameters, but if none is given they all have default values.

Algorithms available include Random Forest (RF), Gradient Boosting, Support Vector Machines (SVM), Logistic Regression (LR), K-nearest neighbours (KNN), Stochastic Gradient Descendent (SGD), Gaussian Naive Bayes (GNB) and Artificial Neural Networks (ANN).

This module allows to perform a grid search or randomized search (with Cross validation) for hyperparameters tuning, returning the best fitted model. For hyperparameter optimization, all the models have default parameter grids but user can specified their own. Allows also for cross validate a model. Within this module, it is also possible to access to the most important features used in the model and plot these results for further analysis.

SVM generates optimal hyperplanes that can maximize the distance between the samples of different classes [3]. The distance to the hyperplane in a higher dimensional space can be computed using several different kernel functions (e.g. linear, rbf). They are powerful methods that work well with sparse data, are less prone to overfitting and work well with both few and many features. Nonetheless, it is necessary to be careful with the pre- processing of data [12, 16]. RF are collections of decision trees where each tree is slightly different from the others. Decision trees are flowchart-like structures that learn a hierarchy of if/else questions, leading to a decision. They don't tend to perform well on high dimensional sparse data but work well on large datasets . Number of decision trees considered and the maximum depth of the tree are important parameters to tune to avoid under and overfitting [12]. LR is a linear model for classification where the probabilities describing the possible outcomes of a single trial are modeled using a logistic function [14]. These models are fast to train and predict, they work well with large datasets and sparse data and are intuitive to understand. With highly correlated features, however, the coefficients can be hard to interpret. Also, they do not model nonlinear relations between the data [12]. SGD is a very efficient and scalable classifier that fits linear classifiers under loss functions such as (linear) Support Vector Machines and Logistic Regression (SGD is an optimization technique and does not correspond to a specific family of machine learning models) [14]. KNN calculates the distances between samples. A query point is assigned the data class which has the most representatives within the nearest neighbors of the point [14]. GNB is an algorithm based on Bayes theorem. These algorithms work by calculating a Gaussian likelihood function between the relative frequencies of each attribute and the frequency of the class variable. Despite of simple and naive assumptions, they have shown to perform well on classification tasks [12]. They are quite similar to the linear models but tend to be even faster in training and predicting, work well with large, high dimensional and sparse data and are relatively robust to parameter choice [12]. ANN implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation. Typical architectures includes the input layer, an hidden layer and the output layer. This implementation is not intended for large-scale applications [14].

This module has a report function, that if activated when initiated the class will store all the results in txt file. The class receives x and y train and test datasets. If no tests dataset are provided only the test scoring functions will not be available.

This class allows to:

- This function performs a parameter grid search or randomizedsearch (change optType) on a selected classifier model and training data set. It returns a scikit-learn pipeline that performs standard scaling (if not None) and contains the best model found by the search according to the Matthews correlation coefficient or other given metric. As models, it is possible to choose between *Random Forest (RF)*, *Gradient Boosting classifier (GB)*, SVM, *k-nearest neighbour algorithms (KNN)*, *stochastic gradient descendent (SGD)*, *gaussian naive bayes (GNB)*, lr and *Artificial neural networks (ANN)*. If param grid are None, the ProPythia default ones will be used.

- Performs cross validations core on a selected classifier model and training data set. It returns the scores across the different folds, means and standard deviations of these scores. The models available are the same as above.

- Retrieve test set scores for the specified scoring metrics allowing for an evaluation of the performance of the model through test-set prediction. It retrieves the values for *Matthews correlation coefficient (MCC)*, accuracy, precision, recall, f1, *receiver operating characteristic (ROC) - Area under curve (AUC)*, *true negative (TN)*, *false positive (FP)*, *false negative (FN)*, *true positive (TP)*, *False discovery rate (FDR)*, sensitivity and specificity derived from metrics *Scikit-learn* module;

- Plot a ROC curve;

- Plot a validation curve for the specified classifier on any parameter given with specific values given in 'param_range';

- Plot a learning curve to determine cross validated training and tests scores for different training set sizes. It retrieves graphic representing learning curves, numbers of training examples, scores on training sets, and scores on tests set and scalability and performance plots if set to True;

- Retrieve the features importances as a dataframe (for classifiers SVM, RF, GB, SGD).For both SVM and SGD the retrieved values are the coefficients of the features and for the RF it is possible to retrieve the values of feature importance;

- Retrieve the features importances explained in previous point represented as barplot.

- Make predictions for unseen sequences (ultimate goal for ML models). The function *predict* can be used to predict novel data with a trained classifier model returning a dataframe with predictions using the specified estimator If the true class is given, the scoring value for the test data is provided;

In the search for the best model, it is possible to adjust several parameters. The pipeline includes the use of a scaler, *standardscaler* by default, but the user, can choose any other from

Scikit-learn such as *normalizer*, *minmaxscale* or None. The score used to find the best model in the grid search can be chosen by the user, being by default the MCC. The MCC is often used in machine learning to measure the quality of binary classifications, being a balanced measure that can be used even if the classes are of different sizes. The values are between 1, a perfect prediction, and -1, total disagreement between predictions and observations. Values of 0 are considered to be no better than random predictions. Sample weights for training data, number of parallel jobs (by default -1) and number of cross validation folds (by default 10) can also be tuned by the user. In all the models available, the parameters can be tuned, however, if none is given, a default parameter grid is set for each model.

A learning curve determines cross validated training and test scores for different training set sizes, being useful to determine if it is beneficial to add more training data and whether the estimator suffers more from a variance error or a bias error (for example: if both validation and training cross converge to a value that is low when increasing the training dataset, the model will not benefit from the addition of more samples).

A cross validation generator splits the dataset k times in training and test data, a score for each training subset and test will be computed and the scores will be averaged over all k runs for each training subset size. The function receives the estimator, title for the graph, train sizes to test and cross validation parameters and retrieves the plot learning curve for that model, numbers of training examples, scores on training sets and scores on test set. It is based on *Scikit-learn* functions.

The ROC curve is a probability curve and is a good way to see how much the model is capable of distinguishing between classes. The higher the AUC, i.e., close to 1, the better the model. The X axis corresponds to the false positive rate and the Y axis to the true positive rate. A larger area under the curve, and the plot on top left corner are ideal. The function *plot_roc_curve* receives a classifier, and test sets and automatically retrieves the plot.

A view of the methods available in this class can be seen in the table 9.

Table 8: Summary table of the methods available in the class `ShallowML`

Class ShallowML	**Aim**	**Methods**
Build model	Parameter grid or randomized search on a selected classifier model	*train_best_model()*
Evaluate model	tests set scores for the specified scoring metrics	score_testset()
	cross validations core on a selected classifier model and training data set	*cross_val_score_model()*
	Function to plot a ROC curve	*plot_roc_curve()*
Model curves	Plots a cross-validation curve for the specified classifier on all tested parameters in a specific param range	*plot_validation_curve()*
	Plot a learning curve to determine cross validated training and test scores for different training set sizes. Obtain also scalability and performance plots	*plot_learning_curve()*
Feature importance	Retrieve the features importances as a dataframe	*features_importances_df()*
	Retrieve the features importances as a plot	*features_importances_plot()*
Predict	Classify novel data with a trained classifier model	*predict()*

3.9 deep_ml

The *deep_ml* module allows to implement supervised DL algorithms for classification, encompassing models using different types of input representation. A DL model is comprised of layers that are chained into a network, mapping the input data to predictions. A loss function compares predictions to the targets producing a loss value used by the optimizer to update the weights of the model in the learning process. Building deep learning models in Keras, used in our implementation, is done by clipping together compatible layers forming data transformation pipelines, being a linear stack of layers the most common approach [4]. Common layers include Dense (fully connected feedforward layers), convolutional, LSTM and embedding layers.

Dense layers perform a linear operation in which every input is connected to every output by a weight. Convolutional and pooling layers are associated with Convnet largely used for image processing; convolutional layers detect local conjunctions of features from the previous, their layer units are organized in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights (filter bank); the pooling layer merge semantically similar features into one, i.e, replace each patch in the input with a single output, which is the maximum or average of the patch value. LSTM layers are used in Recurrent neural networks, seldom applied in Language processing, because of the ability to learn long-term dependencies. RNNs take each vector from a sequence of input vectors and model them one at a time, maintaining in their hidden units a ‘state vector’ that contains information about the history of the past elements of the sequence. Specifically LSTM have an input and output gate and a cell state and forget gate. The gates can regulate the flow of information and allow the network to remember inputs for a long time (that otherwise would vanish) [**ecun2015**].

Baseline models with a very adaptive strategy are provided that can be called and configured automatically with minimum effort. These include the following architectures: DNNs - networks of Dense layers with or without embedding layers; LSTM or BILSTM layers with or without prior embedding and later Dense layers; CNN layers with or without prior embedding and later dense layers; CNN layers, followed by LSTM/BILSTM and Dense layers. Each of this architecture can be configured setting the number of layers of each type and the respective number of neurons. If the user has its own model architecture implemented using Keras, it can be easily plugged-in using the module functions.

The *deep_ml* module includes functions to train and use DL models for prediction, also allowing for the plot of training and validation accuracy. If callbacks for the models are not given, they are generated automatically. The module allows for cross validation and hyper-parameter optimization through randomized and grid search.

This module has a report function, that if activated when initiated the class will store all the results in txt file. The class receives x and y train and test datasets. If no tests dataset are provided only the test scoring functions will not be available. callbacks and model parameters are also given in init function. It is necessary to establish the problem type (binary, multiclass or multilabel) and the number of classes to classify. These parameters are specially important when running baseline models inside Propythia, as it will established the functions inside model and evaluation accordingly.

This class allows to:

- Run DL model. It will train the DL model using callbacks (user specified or internally derived). It will retrieve training and validation accuracy mean and training and validation loss mean. Also gives the performance plots for the model (training and validation accuracy mean and training and validation loss mean) and the summary plot;
- Cross validate DL model;
- Perform gridSearchCV or randomizedSearchCV in DL models;
- Perform model evaluation with model evaluate function. Returns the loss value metrics values for the model in test mode;
- Perform complete model evaluation using multiple scoring functions;
- Plot a Precision-Recall curve. PRC curves summarize the trade-off between the true positive rate and the positive predictive value for a predictive model using different probability thresholds. While ROC curves are appropriate when the observations are balanced between each class, PRC are appropriate for imbalanced datasets;
- Plot receiver operating characteristic (ROC) curve. ROC Curves summarize the trade-off between the true positive rate and false positive rate for a predictive model using different probability thresholds. ROC curves are appropriate when the observations are balanced between each class;
- plots a cross-validation curve for the specified classifier on all tested parameters given a parameter range;
- Plot a learning curve to determine cross validated training and tests scores for different training set sizes. It retrieves graphic representing learning curves, numbers of training examples, scores on training sets, and scores on tests set and scalability and performance plots;

- Predict with a trained classifier model;
- Save model
- Run, Cross validate or do hyperparameter optimization using baseline models:
  - simple dense - Stack of dense layers
  - embedding dense - Embedding layer + Stack of dense layers
  - lstm simple - Stack of lstm/bidirectional layers + stack of dense layers
  - lstm embedding - Embedding layer + Stack of lstm/bidirectional layers + stack of dense layers
  - CNN 1D - Stack of CNN1D (with max pooling) + Stack of dense layers
  - CNN 2D - Stack of CNN2D (with max pooling) + Stack of dense layers
  - hybrid model - Stack of CNN1D (with max pooling) + Stack of lstm/bidirectional layers + Stack of dense layers

A view of the methods available in this class can be seen in the table **??**.

Table 9: Summary table of the methods available in the class `DeepML`

Class DeepML	**Aim**	**Methods**
Build model	Run DL model (train with callbacks)	run_model()
	Perform hyperparameter optimization (grid or randomized Search)	*get_opt_params()*
	Save model	save_model()
	Load saved models	*load_model()*
	Retrieve model inside class	get_model()
Evaluate model	Simple model evaluation returning loss value and metrics values for the model	model_simple_evaluate()
	Complete model evaluation using multiple scoring functions	*model_complete_evaluate()*
	Cross validate DL models	*train_model_cv()*
	Plot a Precision-Recall curve	*precision_recall_curve()*
	Plot ROC curve	*roc_curve()*
Model curves	Plots a cross-validation curve for the specified classifier on all tested parameters in a specific param range	*plot_validation_curve()*
	Plot a learning curve to determine cross validated training and test scores for different training set sizes. Obtain also scalability and performance plots	*plot_learning_curve()*
Predict	Classify novel data with a trained classifier model	*predict()*
Train, CV, HO with Base models	simple dense	*run_dnn_simple()*
	embedding dense	*run_dnn_embedding()*
	lstm simple	*run_lstm_simple()*
	lstm embedding	*run_lstm_embedding()*
	CNN 1D	*run_cnn_1D()*
	CNN 2D	*run_cnn_2D()*
	hybrid model	*run_cnn_lstm()*

3.10 Other functions

Besides these models previously described, the package contains a test directory that contains python file to test all the functions of the package.

It also has an example directory with jupyter notebooks with two case studies using ProPythia functions.

References

[1] Martín Abadi et al. "TensorFlow: Large-scale machine learning on heterogeneous systems". In: (2015). URL: `tensorflow.org`.

[2] Dong Sheng Cao et al. "PyDPI: Freely available python package for chemoinformatics, bioinformatics, and chemogenomics studies". In: *Journal of Chemical Information and Modeling* (2013).

[3] Chih-Wei Hsu, Chih-Chung Chang et al. "A Practical Guide to Support Vector Classification". In: *BJU international* (2008). ISSN: 1464-410X. DOI: `10.1177/02632760022050997`. arXiv: `0-387-31073-8`.

[4] Francois Chollet. *Deep Learning with Python*. 2017. ISBN: 9781937785536. DOI: `citeulike-article-id:10054678`. arXiv: `1-933988-16-9`. URL: `http://www.ncbi.nlm.nih.gov/pubmed/20608803`.

[5] Peter J.A. Cock et al. "Biopython: Freely available Python tools for computational molecular biology and bioinformatics". In: *Bioinformatics* 25.11 (2009), pp. 1422–1423.

[6] Diana Gaspar, A. Salomé Veiga, and Miguel A R B Castanho. "From antimicrobial to anticancer peptides. A review". In: *Frontiers in Microbiology* 4.OCT (2013), pp. 1–16.

[7] Isabelle Guyon and André Elisseef. "An Introduction to Feature Extraction". In: 2006, pp. 1–24.

[8] Zhisong He et al. "Predicting drug-target interaction networks based on functional groups and biological features". In: *PLoS ONE* 5.3 (2010), pp. 1–8.

[9] Thomas Klikauer. "Scikit-learn: Machine Learning in Python". In: *TripleC* (2016).

[10] Ernest Y. Lee et al. *Mapping membrane activity in undiscovered peptide sequence space using machine learning*. Vol. 113. 48. 2016, pp. 13588–13593.

[11] Alex T. Müller et al. "modlAMP: Python for antimicrobial peptides". In: *Bioinformatics (Oxford, England)* 33.17 (2017), pp. 2753–2755.

[12] Andreas C. Muller and Sarah Guido. *Introduction to Machine Learning with Python: A guide for data scientists*. O'Reilly Media, 2017. ISBN: 978-1-4493-6941-5.

[13] Akshara Pande et al. "Computing wide range of protein/peptide features from their sequence and structure". In: (2019), p. 599126.

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[15] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. "A review of feature selection techniques in bioinformatics". In: *Bioinformatics* 23.19 (2007), pp. 2507–2517.

[16] Gisbert Schneider and Uli Fechner. "Advances in the prediction of protein targeting signals". In: *Proteomics* 4.6 (2004), pp. 1571–1580. ISSN: 16159853. DOI: `10.1002/pmic.200300786`.

[17] Arun Sharma et al. "DPABBs: A Novel in silico Approach for Predicting and Designing Anti-biofilm Peptides". In: *Scientific Reports* 6.July 2015 (2016), pp. 1–13.

[18] Lipo Wang, Yaoli Wang, and Qing Chang. "Feature selection methods for big data bioinformatics: A survey from the search perspective". In: *Methods* 111 (2016), pp. 21–31. URL: `http://dx.doi.org/10.1016/j.ymeth.2016.08.014`.