ProPythia - User Guide

Ana Marta Sequeira

February 2020

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

ProPythia is a generic automated platform for the classification of peptides/proteins based on their physicochemical properties and making use of different machine learning models. The package developed facilitates the major tasks of machine learning and it includes modules to read and alter sequences, to calculate protein features, do dataset preprocessing, do feature reduction and selection, perform clustering and to build machine learning models and make predictions. As it is built in a modular way, the user retains the power to manipulate and use others functions outside of the package, having control over the different steps and adapting/extending the code to fit their specific needs. The package is directed to handle protein related problems, but its modular construction allows users to use it in other problems. When compared to the most similar packages, ProPythia is advantageous because:

- 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 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* [9], *PyDPI* [2], *pfeature* [11] and *Biopython* [3] were used.

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. **feature_reduction module**: to reduce the number of features based on the unsupervised technique *Principal component analysis (PCA)*;
5. **feature_selection module**: to discover and select the most valuable features;
6. **clustering module**: to perform and plot the results of a clustering analysis;
7. **machine_learning module**: to implement supervised machine learning algorithms.

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.

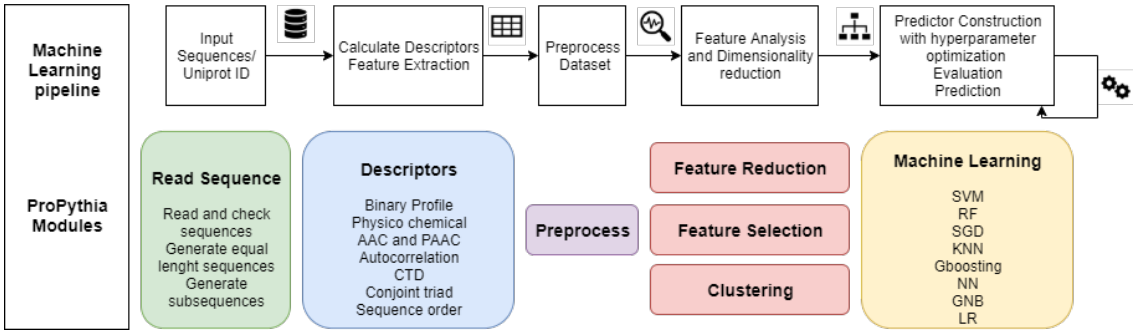

Figure 1: Schematic representation of the modules in the built package

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);
- 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 Equal length	Check if sequence is a valid aminoacid sequence	*checkprotein()*
	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 sub-sequences	*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.

Overall, 38 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

Class Descriptor	Nº	Aim	Method	NºDescriptors
	14	Molar extinction coefficient	*get_molar_extinction_coefficient()*	2
	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()*	-
	20	Aminoacid compositon	*get_aa_comp()*	20
Aminoacid Composition	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()*	-
All	46	Calculate all possible descriptors (except tripeptide)	*get_all()*	-
Adaptable	47	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()*

3.4 feature_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 [10]. This

way, the class `Feature_reduction` enables to reduce the number of features on a dataset based on the unsupervised technique PCA. 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. For better PCA analysis and perception, this module enables the production of two plots. The main purposes of this class are to:

- Derive the PCA of the data. The function receives a dataset, scales it (using standard scaler as default) and applies the *Scikit-learn* PCA algorithm retrieving the fitted and the transformed dataset [4];
- Access the contribution of each feature for the component and the variance ratio of each component;
- Produce a bar plot with percentage of explained variance ratio by components;
- Produce a scatter plot that represents the different classes based on two *Principal component (PC)*s (by default the first two). As the PCA components are orthogonal to each other (not correlated), this usually allows to distinguish classes in a clear way.

It is worth noting that PCA is an 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 `Feature_reduction`

Class FeatureReduction	**Aim**	**Method**
PCA	Perform the PCA analysis	*pca()*
	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()*
	Scatter plot of the labels based on two components (by default the first ones)	*pca_scatter_plot()*

3.5 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 [7]. This selection allows to gain knowledge and understanding of the data, quite important in biological questions [5].

There are three major methods to select features - filter methods, wrapper methods and embedded methods [12] 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 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 univariate and embedded techniques, retrieving a dataframe with features names and scores of univariate tests or select from model 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 5).

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

Class FeatureSelection	**Aim**	**Method**
Feature selection	Univariate feature selector	*univariate()*
	Recursive feature elimination	*recursive_feature_elimination()*
	Select from model	*select_from_model_feature _elimination()*
Retrieve features dataframes	Retrieve a dataframe with features names and scores of importance resulting of the univariate tests and from model selection	*features_scores()*
	Retrieve a dataframe with features names and its ranking position ordered	*rfe_ranking()*

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 [14]. 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.

When the class is initialized, a dataset corresponding to the X dataset (input dataset without the target column) and the target column must be provided. Additionally, the test size to split the data into training and test dataset must be provided (established by default as 0.3).

This class implements methods for:

- Performing K means classification;
- Performing hierarchical clustering;
- Applying the K-means algorithm to train data and predict the test set. These labels can be added as a feature or can replace the previous labels in the dataset. It is possible to see how they perform with the function *classify* [1].

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.

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 [6]. 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 [13]. 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 [8]. The function *hierarchical* takes as input the metric and distance parameters and returns the graphic of the hierarchical division of samples.

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	*kmeans()*
	Perform the kmeans to train data and predict the test set. If add, the labels produced by clustering will be added as features. If replace, labels produced will replace the old labels	*kmeans_predict()*
	Fit the model in train datasets and predict on the test dataset, returning the accuracy	*classify()*
Hierarchical	Perform hierarchical clustering	*hierarchical()*

3.7 machine_learning module

In order to implement supervised machine learning algorithms, the class `Machine_learning` was created. The machine learning module intends to facilitate the application of ML models to the classification of peptides. It is possible to do model selection using grid search for hyperparameters tuning and selecting the best model, do model evaluation, return feature importance (as dataframe or bar plot), and plot validation and learning curves. The user must create an object `Machine_learning` giving the X dataset (inputs) and the column of labels and the size of the test set. The function will load and split the data.

This class allows to:

- Perform a grid search on different model parameters, returning the best fitted model. 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)* and *Artificial neural networks (ANN)*;
- Plot a validation curve for the specified classifier on any parameter in grid search;
- Plot Learning curves;
- 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;
- Retrieve the features importance of the final model for classifiers SVM, RF, GB, SGD. It retrieves a dataframe and draws a bar plot. 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;
- Make predictions for unseen sequences (ultimate goal for ML models). The function *predict* can be used to predict novel peptides with a trained classifier model returning a dataframe with predictions using the specified estimator and test data. If the true class is given, the scoring value for the test data is provided;
- Scan a protein using a sliding window approach to predict sites with a trained classifier model. It will return a dataframe with the subsequences generated, positions in the original sequence, probability of belonging to the class and a class probability if the probability is bigger than 0.99, 0.95, 0.9, 0.8, 0.7, 0.6 or predicted as a negative.

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 7.

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

Class MachineLearning	**Aim**	**Methods**
Build model	Parameter grid search on a selected classifier model and peptide training data set	*train_best_model()*
	Plots a cross-validation curve for the specified classifier on all tested parameters given in the option 'param range'	*plot_validation_curve()*
	Test set scores for the specified scoring metrics in a pandas.DataFrame	*score_testset()*
	Function to plot a ROC curve	*plot_roc_curve()*
	Function that given a classifier retrieves the features importances as a dataset and represent as barplot	*features_importances()*
	Plot a learning curve to determine cross validated training and test scores for different training set sizes	*plot_learning_curve()*
Predict	Predict novel peptides with a trained classifier model	*predict()*
	Scan a protein in a sliding window approach to predict novel peptides with a trained classifier model	*predict_window()*

3.8 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 (based on published articles) using ProPythia functions.

A file *scores.py* (in the directory of the scoring on adjuv_functions) allows to calculate scores for SVM, SVC, RF and GNB comparing two given X datasets and labels considering a 10 fold cross validation. This function can be used as a simple way to compare the results of two datasets for example, when testing methods of feature selection.

References

[1] Dong Sheng Cao, Qing Song Xu, and Yi Zeng Liang. "Propy: A tool to generate various modes of Chou's PseAAC". In: *Bioinformatics* 29.7 (2013), pp. 960–962.

[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] 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.

[4] 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.

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

[6] 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.

[7] Thomas Klikauer. “Scikit-learn: Machine Learning in Python”. In: *TripleC* (2016).

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

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

[10] 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.

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

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

[13] 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.

[14] 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`.