VLfeats.py Documentation

author: Samir Farooq
company: University of Rochester Medical Center
team: Rochester Center for Health Informatics
email: samir_farooq@urmc.rochester.edu or martin_zand@urmc.rochester.edu
last revised: 13 June 2018

This program requires the packages: numpy, colorsys, time, math, operator, pydotplus, webcolors, matplotlib, random, Axes3D, copy, scipy, sklearn, csv, and types. Also requires the two scripts that we wrote for the project corresponding to this one: Centroid.py and Networking.py.

The use of this file is to generate meaningful insights into Viral Load data of patients diagnosed with HIV, particularly in regards to their classification using a hierarchical clustering method on meaningful features, followed by characterizing the results of the clustering with a novel machine learning technique, which we term the centroid algorithm (see centroid documentation for details).

Note that we only write documentation on the functions which are important throughout the VLfeats.py file (often being functions which are versatile; meaning they can be very useful outside the realm of viral load data analysis), all others are left to the end-user to make sense of (if he wishes to do so). For the functions which are mentioned, they are placed in this documentation file in the order that they occur in the script.

Generating Meaningful Features:

`VLfeats.feat_calc (x, y, maximum=7.0, minimum=0.0):`

In this function the user can specify how to manipulate `x` and `y` in order to generate meaningful features. After some research we have developed currently 4 features, found in this function as the variables: `A` (relative area of viral exposure), `aD` (adjusted maximal difference), `rr` (weighted recency reliability), and `IQR` (interquartile range). The user is free to add, remove, or modify these features as he pleases; the rest of the code has been written in a way to handle any amount of features.

`Input:`

`x` : numpy array

Viral load dates in the numerical form (number of days).

`y` : numpy array

Viral load values.

`maximum` : float, (default=7.0)

The maximum viral load value expected; this is not necessarily the same as `max(y)`. Note that while the default is set to 7.0, in later functions we hard-code the maximum to be $\log_{10} 10000010$.

`minimum` : float, (default=0.0)

The minimum viral load value expected; this is not necessarily the same as `min(y)`. Note that while the default is set to 0.0, in later functions we hard-code the minimum to be $\log_{10} 10 = 1$.

`Output:`

`L` : list

A list of viral load features (feature vector).

`VLfeats.get_feat_names ():`

This contains the names of the corresponding features to `feat_calc`.

`VLfeats.get_feat_weights ():`

This returns with what weight the features being returned by `feat_calc`.

`VLfeats.tls (value, base_or_root=2.0, func='log', add1=True):`

Takes either the log or root of the value depending on the parameters.

`Input:`

`value` : integer, float, list, or numpy array

> The value for transformation. If it is a list or numpy array then every value in the array is transformed.

`base_or_root` : string or numeric value, (default=2.0)

> If it is a string, then it must be in the form of 'xn': where 'x' is either 'l' for 'log' or type any other charactr for 'root', and 'n' is the numeric quantity of logging or rooting. E.g. if base_or_root='l10' then the func will be changed to 'log' automatically and base_or_root will be treated as 10.0- hence the output is $log_{10} value$.

`func` : string: either 'log' or 'func'

> Specifies whether we wish to take the log of the value or the root. The input of this parameter is ignored if base_or_root is a string.

`add1` : boolean, (default=True)

> If `add1`=True then we actually add 10 to the value prior to logging or rooting. Otherwise there is no addition.

`Output:`

`new_value` : float or numpy array

> If the original input was an integer or float, then the new transformed value is returned as a float. Otherwise if it was a list or numpy array then the new output will be a numpy array with each value in the array transformed.

`VLfeats.get_Matrix (VLpath, normalize=True, with_order=False, dat=False, tform='l10', days=float('inf'), incomplete=False):`

Using the feature calculation specification from `feat_calc`, the names from `get_feat_names`, the weights from `get_feat_weights`, and the transformation specification from `tls`, this function generates a matrix of values where every column is a feature (in the order given by `feat_calc`) and every row is a patient's feature vector.

`Input:`

`VLpath` : An instance of the class Networking.**Networks**

> Must contain all of the viral load values of the patients, with the primary key referring to the field which the viral loads are found. Also there cannot be any multinode viral loads (see Networking.py documentation).

`normalize` : boolean or string: 'auto' or 'pos only', (default=True)

If False then no normalization takes place. If True or 'auto' then it will check the feature column to see if there are any negative values. If there are negative values then it will be normalized to the range [-0.5,0.5]; where only negative value are mapped to the range [-0.5,0] and only positive values are mapped to the range [0,0.5]. Otherwise, if no negative values are found in the column, the feature is normalized to the range [0,1]. If 'pos only' is inputted, then the column will always be transformed into the range [0,1] regardless of negative values.

`with_order` : boolean, (default=False)

If True then the patient ids are also returned (in the order of the matrix rows). If False, then no such patient ids are returned.

`dat` : boolean, (default=False)

If True then a dictionary of the features is also returned, where the keys are the feature names and the values are the feature values. If False then no such dictionary is returned.

`tform` : string or float, (default='l10')

The value inputted into `tform` will be placed into `tls` under `base_or_root`, which specifies what type of transformation the viral load data should undergo. If the input of `tform` is a numeric value, then it shall be logged with the base of `tform`.

`days` : float, (default=float('inf'))

If the value of days is between [0,1] then during feature calculation, the only viral counts which are used for feature extraction are proportional to the amount of time that the patient has measurements for (e.g. if `days`=0.5 and the patient has measurements going through 2000 days, then only the first 1000 days of measurements are considered for feature extraction). However, if days is any value greater than 1, then this will considered as specifying a day cut-off (e.g. if `days`=100 and the patient has measurements going through 2000 days, then only the first 100 days of measurements are considered for feature extraction). Note that patients will likely not have a measurement right on the `days` specified, hence we choose the measurement which is closest to `days` but not exceeding it (e.g. If `days`=1200 but the patient only has data values for days: 0, 100, 300, 400, and 1500, then feature extraction is only performed on the measurements corresponding to the days: 0, 100, 300, and 400). Also note that the patient is only considered for feature extraction if the total number of measurements is $\geq$3, otherwise the patient is skipped during the feature extraction step.

`incomplete` : boolean, (default=False)

If `incomplete`=True and if `days` is also >1, then we skip patients whose measurements do not span the number of `days`. E.g. If days=1000, but a patient only has measurements spanning 100 days, then this patient will be skipped from feature extraction if `incomplete`=True. On the other hand, if a patient had measurements spanning 1200 days, then this patient will be included in the feature extraction step in the way described under `days`. If `incomplete`=False, then there is no such skipping that takes place.

`Output:`

`M` : numpy array, shape = [n_patients_considered, n_features]

A matrix whose columns are the features corresponding to the output of `calc_feat`. Note that not all patients in `VLpath` will necessarily found in `M`, because we skip patients who do not have $\geq$3 viral load measurements.

`data` : dictionary, (not returned by default)

This is only returned with `M` if `dat`=True, in which case it will returned as the second item in the tuple. This dictionary has as its keys the feature names specified by `get_feat_names` and has as its values a list of feature values (the amount of n_patients_considered).

`p_ids` : list, (not returned by default)

A list of the patients whom were considered. This is only returned if `with_order`=True, in which case it will be the third item of the tuple being returned if `dat` also equals True, otherwise it will be the second item being returned.

Hierarchical Clustering

class `VLfeats.linkage_clustering (Z, thresh, label=None):`

This class has been designed to be more versatile than scipy.cluster.hierarchy.**dendrogram** and provide more meaningful results to the end-user; which is to use linkage clustering results to create cluster structures based on a provided threshold. Note that running `linkage_clustering` will act to perform the structuring of the clusters, but will not plot anything. For plotting please either first use the method `generate_cluster_colors` if you wish to assign specific color the classes, then/or use the method `plot_dendrogram` (which does not need `generate_cluster_colors` to have already been run to operate).

`Parameters:`

`Z` : numpy array, shape = [n_patients - 1, 4]

This numpy array should come from the result of performing scipy.cluster.hierarchy.**linkage** on a matrix (i.e. the matrix outputted from `get_Matrix`).

`thresh` : float

The specific threshold to make a cut on a hierarchical dendrogram.

`label` : numpy array or None, (default=None)

These are the labels of the rows (patients if working with viral load data). If `label`=None then any subsequent plots generated from this class structure will not be labeled on the x-axis.

`Attributes:`

`Clusters` : list

A list of sets, where the sets contain the patients (rows) belonging to the same cluster. Hence the length of `Clusters` is equal to the number of clusters.

`colors` : dictionary

Does not exist as an attribute until either `generate_cluster_colors` is run or `plot_dendrogram` is run. The keys are the cluster numbers (corresponding to the index of list `Clusters`), and the values are color names or RGB color codes.

`def_bracket_color` : numpy array, shape = [3,]

Does not exist as an attribute until either `generate_cluster_colors` is run or `plot_dendrogram` is run. This is the color which the brackets above the cutting threshold will have. While there is no method to change this color, one can change it manually by first running `generate_cluster_colors` followed by manually changing this attribute to the desired color.

`Z` : numpy array, shape = [n_patients - 1, 4]

The same `Z` which was inputted as the parameter.

`thresh` : float

The same `thresh` which was inputted as the parameter.

`label` : list

The labels which were inputted.

`Zt` : numpy array

This `Z` with the threshold applied to it.

`B` : dictionary

Contains a reference to each patient (or row) to which cluster number the patient belongs. The cluster number is the same as the index number of the for the `Clusters` attribute.

`p` : integer

The number of patients (rows).

`cn` : integer

Same as `len(Clusters)`.

`S` : set

Record keeper, not to be touched by end-user.

`C` : set

Record keeper, not to be touched by end-user.

Methods:

`generate_cluster_colors (colorblind_friendly=True, set_own=False)`

If colorblind_friendly=True, then there cannot be any more than 8 clusters. These colors are taken from http://mkweb.bcgsc.ca/colorblind/. If one wishes to set their own colors then they can do so by first setting colorblind_friendly=False, then changing set_own to a dictionary: where the keys are the cluster numbers (corresponding to the index of `Clusters`), and the values are the color names or RGB color codes. If both inputs are false then we draw equally distributed colors from the hue color-map.

`plot_dendrogram (ax=None)`

Plots the dendrogram on a new figure if no ax is provided, otherwise plots the dendrogram on ax. To turn on or off labels on x-axis or to adjust the cut threshold, then one will have to re-run `linkage_clustering` from scratch.

`add_subcomponent (i, cluster)`

This is a back-end method which is not meant for the user to play around with- responsible for structuring the patients together.

`plot_segment (x, bot, top, color)`

This is a sub-routine of `plot_dendrogram` and should not be touched by user.

`check_position (index)`

This is a sub-routine of `plot_dendrogram` and should not be touched by user.

`update_bracket (zi)`

This is a sub-routine of `plot_dendrogram` and should not be touched by user.

`VLfeats.auto_cluster (Net, title='Dendrogram', cluster_threshold=3.8, update_class=False, show_feat_plot=False, transform='l10', include_heatmap=False, OT=True, lw=0.5, fs=(15.5,8.5), save=False):`

Performs hierarchical clustering on patients. It uses `Net` as a call to `get_Matrix` to retrieve `M`, from which we apply scipy.cluster.hierarchy.linkage to `M` to attain `Z`. With `Z` and the `cluster_threshold` we call `linkage_clustering` to attain the clustering results of the patients.

`Input:`

`Net` : Networking.**Networks**

Object of Networking.**Networks** containing viral load information as its `primary_key` (see Networking.py documentation for details).

`title` : string, (default='Dendrogram')

Title for the dendrogram plot.

`cluster_threshold` : float, (default=3.8)

The hierarchical clustering cutting threshold.

`update_class` : boolean or 'only', (default=False)

Whether or not to update classes found from the clustering results. If 'only' is inputted then no plotting occurs.

`show_feat_plot` : boolean, (default=False)

Whether or not to show the bivariate scatter plot with cluster colors.

`transform` : string, (default='l10')

How to transform the viral load data. Default is to add 10 and then take the log with base 10.

`include_heatmap` : boolean, (default=False)

Whether or not to include the heatmap underneath the dendrogram plot.

`OT` : boolean, (default=True)

How to normalize the columns of the matrix `M`- this is a call to the `normalize` parameter in `get_Matrix`.

`lw` : float

The line width of the dendrogram brackets

`fs` : tuple, (default=(15.5,8.5))

The size of the figure that is generated

`save` : False or string, (default=False)

If False then the generated figure is not saved. To save the figure enter a string to name the file (including the extension of your choice).

`Output:`

`Net` : Networking.**Networks**

If `update_class` is True or ‘only’ then returns the updated viral load network with the new class information. Otherwise outputs the same `Net` as was inputted. Note: to add meaningful cluster names to the Networks object then refer to the `rename_classes` function in the Networking.py documentation.

Categorization Method Comparison

`VLfeats.my_cmaps (M, my_cmap=‘pink’):`

Returns a colormap (and a complementary color depending on the input). A few of these are custom built colormaps which we describe below.

`Input:`

`M` : numpy array

While an array is not technically necessary for a colormap, some of our custom built colormaps are sensitive to the second minimum value of an array of interest (this is useful for when the distinction between the value of 0, and the next minimum value needs to be emphasized in the coloring). In the list of colormap options, if you see ‘crisp’ in parenthesis, then know that the first color in the sequence is meant for values of 0. If these custom colormaps do not interest you than you can input any list of numbers and it will not effect the color map.

`my_cmap` : string, (default=‘pink’)

Our custom colormap names are listed below (the term ‘crisp’ means it attempts to differentiate values of 0 and the rest of the values by making the first color in the sequence unique to 0 values):

- ‘pink’: Cyan → White → Pink (crisp)
- ‘cool’: White → Cyan → Pink (crisp)
- ‘lightautumn’: Light Cyan → Very Light Yellow → Light Red/Salmon (crisp)
- ‘autumn’: Light Cyan → Light Yellow → Light Red/Salmon
- ‘teal’: Copper → White → Teal (crisp)
- ‘lightpurple’: Light Cyan → Very Light Pink → Light Purple

`Output:`

`cmap` : color map, (not returned by default)

One of the color maps as listed above.

`crgb` : list- RGB code

This is only returned if the input for `my_cmap` has a comma contained in the string, with the first name before the comma being the colormap of interest and the name after the comma (no space) is the complementary RGB code of interest (this is used for the scatter plotting in `binned_time_series`. Possible options for this complementary color are: ‘green’, ‘black’, ‘lime’, ‘lightviolet’, ‘lightcyan’, ‘red’, ‘mgold’, ‘copper’, ‘bloodred’, and ‘brown’.

`VLfeats.binned_time_series (T, Y, ax, mint=0.0, maxt=1863.2, miny=1.0, maxy=7.0, matrix_shape=(12,28), xtl=identity, ytl=identity, xts=14, yts=14, halfxticks=False, halfyticks=False, plot_last=False, cmap=‘default’)):`

This function creates a binned time series plot. In our case we use it for viral load data but can accept any type of time-series data as long as it is coherent to the input rules.

`Input:`

`T` : list or numpy array

> Numeric time points for the time-series plotting. Must be a list of lists (or a numpy array) where each row is a different time series (patient), and each list within the row is of the time points for that particular time-series (not necessarily of the same lengths). E.g.
>
> [[0,2,5,8,15],
> [0,10,28],
> [0,6,12,19,23,35]]
>
> In this example there are three distinct time series, all with different number of time instances (which is not required).

`Y` : list or numpy array

> Takes the same format at T, but this refers to the dependent variable that is of interest (viral load data in our case).

`ax` : An object of: matplotlib.axes._subplots.AxesSubplot

> The subplot (ax) to plot the binned time-series on is required.

`mint` : float, (default=0.0)

> The minimum time point to consider for `T`. This must be equal to or below the true minimum of `T` (i.e. `mint`$\leq$`numpy.min(T)`).

`maxt` : float, (default=1863.2)

> The maximum time limit for the binned plot. If there are any time values in `T` which are greater than `maxt`, then it will simply be binned into the last bin in regards to time (x-axis).

`miny` : float, (default=1.0)

> The minimum dependent value (viral load value) to consider for `Y`. This must be equal to or below the true minimum of `Y` (i.e. `miny`$\leq$`numpy.min(Y)`)

`maxy` : float, (default=7.0)

> The maximum dependent value for the binned plot. If there are any dependent values in `Y` which are greater than `maxy`, then they will be binned into the last bin in regards to the dependent (y-axis).

`matrix_shape` : tuple, (default=(12,28))

> The shape of the binned time series plot. Inclusive in this argument is the number of bins one wishes to have.

`xtl` : function, (default=lambda x:x)

> Stands for 'x tick labels', which specifies how you wish to transform the xtick label values. The transformation is inputted as a function and by default there is no transformation (i.e. the function returns itself).

`ytl` : function, (default=lambda x:x)

Stands for 'y tick labels', which specifies how you wish to transform the ytick label values. The transformation is inputted as a function and by default there is no transformation (i.e. the function returns itself). This is useful for when the y-tick labels should be changed in accordance to reverse logging (for clearer plotting), which we have made the `rev_log10_p10` function for.

`xts` : integer, (default=14)

font-size for x-tick labels

`yts` : integer, (default=14)

font-size for y-tick labels.

`halfxticks` : boolean, (default=False)

Sometimes the xtick labels can be very close/squished together, making the labels hard to read. If this value is set to True, then only half of the xtick labels are kept.

`halfyticks` : boolean, (default=False)

Sometimes the ytick labels can be very close/squished together, making the labels hard to read. If this value is set to True, then only half of the xtick labels are kept.

`plot_last` : boolean, (default=False)

Whether or not to plot the last time series point of each time-series on the binned time-series plot in the form of a scatter.

`cmap` : color map, (default='default')

The color map to use for the binned time-series plot. If `cmap` is not equal to 'default', then `my_cmaps` is called using `my_cmap=cmap` (so see the details of this function for input options; if this route is taken, then you must have `crgb` also as an output of `my_cmaps`). If none of the coloring options interest you then you will need to manually add your own to the function.

Classification Stability

`VLfeats.`**MLmodel** `(alg='DecisionTree', trees=150, w=get_feat_weights())):`

This function makes it easy to call different supervised machine learning algorithm implementations in python.

`Input:`

`alg` : string, (default='DecisionTree')

The choices are listed below. Note that for areas where '_' is found in the string, the user is not supposed to type '_' but rather replace it with a parameter of choice, which is explained:

- 'Logistic': Logistic Regression
- 'RF': Random Forests
- 'SVC': Support Vector Machine (SVM)
- 'LDA': Linear Discriminant Analysis
- 'QDA': Quadratic Discriminant Analysis
- 'kNN_': k-Nearest Neighbors, choose k by picking a specific number in replace of '_'.
- 'AdaBoost': Adaboost
- 'DecisionTree_': Decision Tree. _ controls the maximum depth allowed in the decision tree. Thus _ should either be an integer or be left blank if no maximum is desired.

* ‘NeuralNet’: Backpropogation algorithm.
* ‘Centroid _ _’: Calls Centroid.py, where the first ‘_’ is the center detection method of choice, and the second ‘_’ is the prediction method of choice. The space between the two is essential. For parameter choices refer to the Centroid.py documentation.

`trees` : integer, (default=150)

This is used for additional parameters. This number is used as the number of estimators for random forests, and the number of estimators for adaboost. Also this number multiplied by 50 is used for the max_iterations allowed for backpropogation. For all other algorithms, this parameter is ignored.

`w` : list, (default=`get_feat_weights()`)

This is a list of how to weight the features. This is only used in the Centroid model.

`Output:`

`MLmodel` : varying types

The supervised machine learning model is returned. All of the above models are scikit-learn implementations, with the exception of the Centroid model, but we have coded that model to be very similar to scikit-learn.

`VLfeats.get_training_data (VLpath, return_dummy=False, upsample=False):`

Retrieves training data for our supervised machine learning algorithms to learn on, extracted similarly to `get_Matrix` with a few important distinctions. The main distinction is that we get target (class) labels to learn on, and a second important distinction is that it allows to upsample if you wish.

`Input:`

`VLpath` : Networking.**Networks**

An object of Networking.**Networks** which contains viral load data in its `primary_field` (see Networking.py documentation for details).

`return_dummy` : boolean, (default=False)

Whether to return the target (class) labels as dummy numbers. If this is chosen then we also return the outputs `dummy_c` and `T` along with the default outputs. `dummy_c` is a list of dummy class targets (i.e. the classes are still grouped the same, but replaced with dummy numbers). `T` is a dictionary that identifies which class the dummy numbers actually refer to and vice versa.

`upsample` : False or integer, (default=False)

If you wish to upsample/downsample data (because sometimes there are rare classes that need to be treated as more important or vice versa) then specify the amount of samples that you wish to have. For classes with more than the samples you inputted they will be downsampled without replacement, and for classes with less than the samples inputed then they will be upsampled with replacement.

`Output:`

`M` : numpy array, shape = [n_patients_considered, n_features]

Training data.

`true_c` : numpy array, shape = [n_patients_considered,] Class (target) data.

`dummy_c` : numpy array, shape = [n_patients_considered,], (not returned by default)

Only outputted if `return_dummy`=True. This contains class (target) data but all the real target names are replaced with dummy integers.

`T` : dictionary, (not returned by default)

Only outputted if `return_dummy`=True. This dictionary identifies which class the dummy numbers actually refer to and vice versa.

`VLfeats.trainingVL (VLpath, alg='kNN7', plot=False, use_LOOCV=False, given=False):`

Measures performance of machine learning algorithms on the viral load data by finding F_1 scores associated with prediction.

Input:

`VLpath` : Object of the class Networking.**Networks**

An object of Networking.**Networks** which contains viral load data in its `primary_field` (see Networking.py documentation for details).

`alg` : string, (default='kNN7')

The supervised learning algorithm to test performance of. Choices of the algorithms are mentioned in `MLmodel`.

`plot` : boolean, (default=False)

Whether or not to generate a plot showing the prediction accuracy. In the plot, if you choose to do so, the 'x' represents mislabeled patients (with the color referring to the label the predictor predicted it as), and '.' are the correctly labeled patients.

`use_LOOCV` : boolean, (default=False)

If False then performance (prediction) is measured on the same set it trains on (this is much faster but gives less reliable answers). If True then uses Leave One Out Cross Validation of the training set to test performance (this is much slower, but gives more reliable answers).

`given` : False or tuple, (default=False)

If `M, true_c, dummy_c,` and `T` are already known, then input as a tuple for this parameter in this order. Otherwise they will be calculated from `get_training_data`.

Output:

`scores` : dictionary

Performance results in the form of a dictionary: the keys are the class labels and the values are a list of 4 numerical numbers: *ROC score, precision, recall,* and F_1.

`VLfeats.stacked_barplot (x, y, ax, C, thisC, include_thisC=False, avg=False, title=True):`

Generates a sequential stacked barplot.

Input:

`x` : list or numpy array

Contains the independent variable that we are measuring the stacked bar plots over (this could be time; in our case it was used for proportional time).

`y` : list

Must be a list of dictionaries. The length of the list must be equivalent to that of `x`. For each dictionary: the keys must be the names of the classes of interest, and its values should be the count/occurrence of that class respective to the value of `x` (in terms of index).

`ax` : An object of: matplotlib.axes._subplots.AxesSubplot

The subplot (ax) to plot the sequential stacked barplot on is required.

`C` : list

The names of all the classes. This is included for the sake of retaining class plotting order.

`thisC` : Any hashable object

The name of the class which is of particular focus. If there is no class of particular focus the simply put in any class name here and switch `include_thisC` to True. This parameter is useful for looking at class prediction performance.

`include_thisC` : boolean, (default=False)

Whether or not to include `thisC` in the stacked barplot.

`avg` : boolean, (default=False)

Whether to plot based on absolute counts/occurrences or to plot as a relative measure (i.e. averaging the counts to be between [0,1]).

`title` : boolean or string, (default=True)

If False no title is included. If True then the title is set to the value of `thisC`. Otherwise if the user inputs a string then the title will be whatever the user inputted.

`Output:`

`ax` : An object of: matplotlib.axes._subplots.AxesSubplot

The same axis which the user inputted, only modified with the stacked barplots

`total_pats` : list

A list of the sum of all the counts if `avg`=False, otherwise it will be a list of ones.

`VLfeats.validatingVL (VLpath, disp='auc', alg='kNN7', upsmp=False, step=15, mxdays=1876, specific_save_name=False, return_scores=False):`

Generates performance scores for each class label and plots the classification stability results. It is labeled 'validating' but since writing the function name we no longer believe that this is any form of 'validation', but rather it is a form of testing stability of our classes.

`Input:`

`VLpath` : Object of the class Networking.**Networks**

An object of Networking.**Networks** which contains viral load data in its `primary_field` (see Networking.py documentation for details).

`disp` : string, (default='auc')

Determines what measure we want to visualize. If `disp`='auc', then the AUROC (area under the ROC curve) of each class is plotted. If the user to instead plot the sequential stacked barplot, then the user may type anything else: however the algorithm will look for certain key words such as 'abs' and 'avg'. If it detects that 'avg' exists in `disp` then it will average the counts (see `stacked_barplot`), and/or if it detects that 'abs' exists in `disp` then it will count the predicted class as an absolute measure (e.g. either a patient is classified as class A or it is not; this comes from the `.predict` method of the supervised learning algorithm), otherwise the predicted class will be scored as a proportional measure (e.g. a patient is 55% class A, 35% class B, and 10% class C; this comes from the `.predict_proba` method of the supervised learning algorithm).

`alg` : string, (default='kNN7')

Please see `MLmodel` for options regarding this parameter.

`upsmp` : boolean, (default=False)

Please see `upsample` from `get_training_data` for details of this parameter.

`step` : integer, (default=15)

If the user wishes to observe the classification stability in terms of days since first viral load measurement, then this parameter controls in what step should we measure the stability (i.e. days between each stability observation).

`mxdays` : integer or 'prop', (default=1876)

If this quantity is an integer, then the observation will be in days since first viral load measurement. Otherwise if `mxdays`='prop', then the parameter `steps` will be ignored and we will observe the stacked bar plot in terms of relative (or proportional) days (i.e. a proportion range between [0,1] where the range is the proportion of days covered relative to the final measurement date). In this case we use `numpy.linspace(0, 1, 1001)` as our interval of values.

`specific_save_name` : False or string, (default=False)

If False then the figure will save with the name 'Class Validation with'... with the rest of that string being filled in with the algorithm and display type (in the form of a pdf). Otherwise if the user desires their own save name, then they may input it here (including the extension of their choice).

`return_scores` : boolean or 'only', (default=False)

If 'only' is inputted, then the function will only return the score results, otherwise it will return the scores only if the input is True.

`Output:`

`scores` : list, (not returned by default)

A list of lists: where the inside lists are the necessary input results for `stacked_barplot` (see this function for more detail), where the lists are ordered by the integer order of `dummy_c` from the result of `get_training_data` (the order is gauranteed to be held the same as long as the classes in `VLpath` are not modified; `score` is of this format only if `disp` is not 'auc'). If `disp`='auc' then `score` is a list of lists whose inner lists contain results from sklearn.metrics.roc_auc_score.

`VLfeats.wilcoxon_test (VLpath, scores, score_names, save=False, save_table=False):`

Performs Wilcoxon signed rank test on the difference between the algorithms in question. Also plots a boxplot result of the differences.

Input:

`VLpath` : Object of the class Networking.**Networks**

An object of Networking.**Networks** which contains viral load data in its `primary_field` (see Networking.py documentation for details).

`scores` : list

A list of score results from `validatingVL` for each algorithm that we are interested in. Note that the first score (first index of `scores`; i.e. `scores[0]`) must be the algorithm that we are comparing all the other algorithms against.

`score_names` : list

list of strings, whose strings are names of the algorithms (used for plotting).

`save` : False or string, (default=False)

If False then the figure is not saved. Otherwise enter the desired filename in the form of a string (including the extension of choice).

`save_table` : boolean, (default=False)

If True then it will save a table of the IQR results of the difference between the algorithms in comparison the the first inputted algorithm (it will be saved under the name 'Table S2.csv'. If False then no table is saved.