fuzzy-rough-learn 0.2: a Python library for fuzzy rough set algorithms and one-class classification

Oliver Urs Lenz*, Chris Cornelis*, Daniel Peralta†
*Computational Web Intelligence
Department of Applied Mathematics, Computer Science and Statistics
Ghent University, Ghent, Belgium
Email: {oliver.lenz,chris.cornelis}@ugent.be
†IDLab
Department of Information Technology
Ghent University – imec, Ghent, Belgium
Email: daniel.peralta@ugent.be

***Abstract*—We have expanded the scope of fuzzy-rough-learn 0.2 from fuzzy rough sets to also cover one-class classification, facilitating the exploration of practical and conceptual connections between these two areas of machine learning. The new algorithms for one-class classification consist of nine data descriptors and one feature preprocessor. In addition, we have added Fuzzy Rough Nearest Neighbour regression and a number of preprocessors for feature scaling. Apart from these new core algorithms, we have rewritten fuzzy-rough-learn from the ground up, and included a large number of utility functions, to allow users to easily adapt any of the algorithms to their use case.**

I. Introduction

fuzzy-rough-learn is a Python library for machine learning algorithms based on fuzzy rough sets. Version 0.1 of fuzzy-rough-learn has been documented in [1]. This paper is devoted to version 0.2, which is greatly expanded.

The principal goal of fuzzy-rough-learn is to make promising soft computing algorithms from the literature available to other researchers, both to enable their application to practical problems and to facilitate empirical comparisons and further development. fuzzy-rough-learn is inspired by scikit-learn [2] and relies on some of its algorithms as a backend. scikit-learn is one of the most popular Python libraries for general machine learning, and only admits "well-established algorithms" with at least 200 citations[1], a criterion not (yet) satisfied by most algorithms in fuzzy-rough-learn.

The new content of fuzzy-rough-learn 0.2 consists, broadly speaking, of four parts:

1) We have expanded the scope of fuzzy-rough-learn to also cover one-class classification [3], also known as novelty detection or semi-supervised outlier or anomaly detection. One-class classification algorithms, known as data descriptors, can be seen as providing a generalisation of a finite set (expressing a concept) to a fuzzy set on the entire attribute space. In this sense, they are conceptually similar to the upper approximation of a fuzzy rough set, which encapsulates all instances possibly belonging to a concept. Conversely, it is possible to implement both the upper and lower approximations of a fuzzy rough set using data descriptors. By implementing these types of algorithms together in one same software library, we hope to facilitate the exploration of these connections. We have included nine data descriptors, as well as one feature preprocessor (SAE [4]) that was specifically designed to increase the performance of data descriptors.
2) The core component of fuzzy rough set algorithms has been expanded by including a first regression algorithm (FRNN [5]), and expanding the options for FROVOCO classification.
3) We have added a convenient way to incorporate feature preprocessing (including feature scaling) with the different algorithms.
4) We have redesigned the fuzzy-rough-learn from the ground up, to make it even easier for users to control the behaviour of the included algorithms and supplement their own alternatives.

In Section II, we briefly go over the definitions of fuzzy rough sets and one-class classification. Next, we describe some of the formal properties of fuzzy-rough-learn in Section III and outline our design principles in Section IV. Then in Section V we list the core algorithms of fuzzy-rough-learn, and in Section VI, the utility functions. Finally, in Section VII, we briefly outline the future direction of work on fuzzy-rough-learn.

II. Theoretical background

In this section, we give brief definitions of soft maxima and minima, fuzzy rough sets, and data descriptors. At present, we assume that all datasets are numerical, and can thus be seen as a multisubset of $\mathbb{R}^m$, for some integer m.

Definition 1. Let X be a finite collection of values in $\mathbb{R}$ of size n, and let w be a weight vector of length $k \leq n$, with values in $[0, 1]$ that sum to 1. Then the soft maximum and minimum of X induced by w are defined as follows:

[1] https://scikit-learn.org/stable/faq.html

$$\text{soft max}\, X := \sum_{i \le k} w_i \cdot x_i^+; \tag{1}$$

$$\text{soft min}\, X := \sum_{i \le k} w_i \cdot x_i^-, \tag{2}$$

where x_i^+ and x_i^- are, respectively, the ith largest and ith smallest values of X.

The soft maximum becomes an ordered weighted averaging (OWA) operator [6] if we pad the weight vector w with zeros to length n, while the soft minimum corresponds to the OWA operator equipped with the dual of this padded weight vector.

Definition 2. Let $X \subset \mathbb{R}^m$ be a dataset, let R be a fuzzy tolerance relation on $\mathbb{R}^m$, and let T be a choice of t-norm, I a choice of fuzzy implication and w_1 and w_2 choices of weight vectors for the soft maximum and minimum. Then for any fuzzy set C in X, the upper approximation $\overline{C}$ and lower approximation $\underline{C}$ of C are the fuzzy sets in $\mathbb{R}^m$ defined as follows:

$$\begin{aligned} \overline{C}(y) &= \operatorname*{soft\,max}_{x \in X} T(R(y,x), C(x)) \\ \underline{C}(y) &= \operatorname*{soft\,min}_{x \in X} I(R(y,x), C(x)) \end{aligned} \tag{3}$$

We say that upper and lower approximations are *strict* if w_1 and w_2 have length 1 and induce the normal maximum and minimum. Note that when C is crisp, $\overline{C}(y)$ simplifies to $\text{soft max}_{x \in C} R(y,x)$, and $\underline{C}(y)$ to $\text{soft min}_{x \in X \setminus C} N_I(R(y,x))$, where N_I is the fuzzy negation induced by I, defined by $N_I(x) := I(x,0)$.

Definition 3. A data descriptor is a machine learning algorithm that takes a dataset $C \subset \mathbb{R}^m$ for some integer m, and returns a fuzzy set in $\mathbb{R}^m$, its *model* trained on C.

The model generated by a data descriptor is a generalisation of C to $\mathbb{R}^m$, and can be used to predict membership of C for new instances in $\mathbb{R}^m$ (one-class classification). The link between fuzzy rough set theory and one-class classification is that the upper and lower approximation constructions can be seen respectively as a data descriptor applied to C, and the negation of a data descriptor applied to $X \setminus C$.

III. Formal specifications

fuzzy-rough-learn is hosted on the two principal repositories for Python libraries, pipy and conda-forge, and thus can easily be installed with either `pip` or `conda`. Its documentation is located at https://fuzzy-rough-learn.readthedocs.io. The source code is accessible on GitHub at https://github.com/oulenz/fuzzy-rough-learn, which also offers users the opportunity to report issues or contribute improvements or additional material. fuzzy-rough-learn is distributed under the MIT license [7], making it freely usable for any purpose.

fuzzy-rough-learn 0.2 requires Python 3.7 or later. Its dependencies are NumPy 1.17, SciPy 1.1 and scikit-learn 0.22. In addition, use of the EIF data descriptor requires the eif library, and the SAE feature preprocessor requires TensorFlow and Keras.

TABLE I
Internal subdivision of fuzzy-rough-learn.

Name	Description
`neighbours`	Nearest neighbour algorithms
`networks`	Neural networks
`statistics`	Statistical functions
`support_vectors`	Support vector machines
`trees`	Decision trees
`uncategorised`	Other functions

IV. Design principles

The two primary design principles of fuzzy-rough-learn are consistency and modularity. To achieve this, we have chosen a uniform structure based on NumPy arrays and functions. As in scikit-learn, datasets are n by m two-dimensional arrays, where n is the number of records, and m the number of features. Machine learning models are functions that take a (test) dataset and return one or more values for each record. This means that machine learning *algorithms* are second-order functions, which take a (training) dataset and return a model. An algorithm is supervised if, in addition to a dataset, it also takes an array of labels.

fuzzy-rough-learn uses classes, but in a restricted way. At present, the classes only possess two user-facing methods: the initialisation method `__init__`, and the call method `__call__`. Initialisation allows users to set hyperparameter values. Thanks to `__call__`, an object of this class, once initialised, behaves just like a function.[2] Therefore, we call this a parametrisable function. To make it clear for users whether a function is a parametrisable function that must be initialised, we use `CamelCase` if this is the case, and `snake_case` if not (in line with the Python convention for class and function names).

Conceptually, the functions in fuzzy-rough-learn can be subdivided into core algorithms and utility functions (Fig. 1, sections V and VI). The source code is organised into domains of machine learning (Table I). However, end users are presented with a flat hierarchy, which only groups functions according to their functionality (the groups in Fig. 1). Thus, e.g. the FRNN classifier can be imported with `from frlearn.classifiers import FRNN`.

The advantage of the design pattern outlined in the previous paragraphs is that it offers users a large amount of flexibility. They can control the principal design choices of each algorithm by setting one or more hyperparameter values. If a user requires greater flexibility, they can substitute their own class by inheriting from one of the abstract base classes. And finally, thanks to the concept of duck typing in Python, they can also substitute any simple function that accepts the relevant input and produces the required output.

[2]To use the technical term, it is a 'callable'. Classes themselves are also callables, and many so-called functions in the Python standard library are in fact classes.

Core algorithms

Classifiers	Data Descriptors	Feature Preprocessors	Instance Preprocessors	Regressors
FRNN*	ALP	FRFS*	FRPS*	FRNN
FRONEC*	CD	LinearNormaliser		
FROVOCO*	EIF‡	IQRNormaliser		
	IF†	MaxAbsNormaliser		
	LNND	RangeNormaliser		
	LOF	Standardiser		
	MD	SAE		
	NND	VectorSizeNormaliser		
	SVM†			

Utility functions

Array functions	Dispersion measures	Location measures	Neighbour search methods	Parametrisations
div_or	interquartile_range	maximum	BallTree†	log_multiple
first	maximum_absolute_value	mean	KDTree†	multiple
greatest	standard_deviation	median		
last	total_range	midhinge		
least		midrange		
remove_diagonal		minimum		
soft_head				
soft_max				
soft_min				
soft_tail				

T-norms	Transformations	Vector size measures	Weights	Other (postprocessing)
goguen_t_norm	contract	MinkowskiSize	ConstantWeights	discretise
heyting_t_norm	shifted_reciprocal		ExponentialWeights	probabilities_from_scores
lukasiewicz_t_norm	truncated_complement		LinearWeights	select_class
			QuantifierWeights	
			ReciprocallyLinearWeights	

Fig. 1. Schematic overview of the contents of fuzzy-rough-learn. *Documented in [1]. †Wrapper for implementation in scikit-learn. ‡Wrapper for implementation in eif library.

Among the algorithms, feature preprocessors play a special role, since they are not generally used on their own, but in combination with one of the other algorithms. For users, this can be impractical, since it is not enough to apply each feature preprocessor on the training data to construct the respective models, but each model then has to be applied to transform the training data, as well as any and all test data, in the correct order. Therefore, we automate this process and allow users to supply any number of feature preprocessors as a hyperparameter when initialising any algorithm. We also use this functionality to equip some algorithms with default feature preprocessors, which may be overridden by the user if so desired.

V. Core algorithms

In this section we discuss the main algorithms included in fuzzy-rough-learn. As a general rule and where applicable, users can set the dissimilarity measure, the number of nearest neighbours, the weights used for aggregation, and the nearest neighbour search algorithm.

In addition to the newly implemented algorithms described below, we have made the FROVOCO algorithm [8] more flexible by adding hyperparameters which control the imbalance ratio that determines which classification subtasks are considered imbalanced, as well as the weights that are used for balanced and imbalanced subtasks.

A. Data descriptors

Fig. 2 illustrates the models obtained from applying the data descriptors in fuzzy-rough-learn on the same toy dataset. ALP, LNND, LOF, NND and SVM have default hyperparameter values that were found to be optimal in [9].

1) ALP: Average Localised Proximity [9]. Localises the nearest neighbour distance of a test instance y against the local nearest neighbour distances in the target data.

The local ith nearest neighbour distance $D_i(y)$ relative to y is the weighted average of the ith nearest neighbour distances

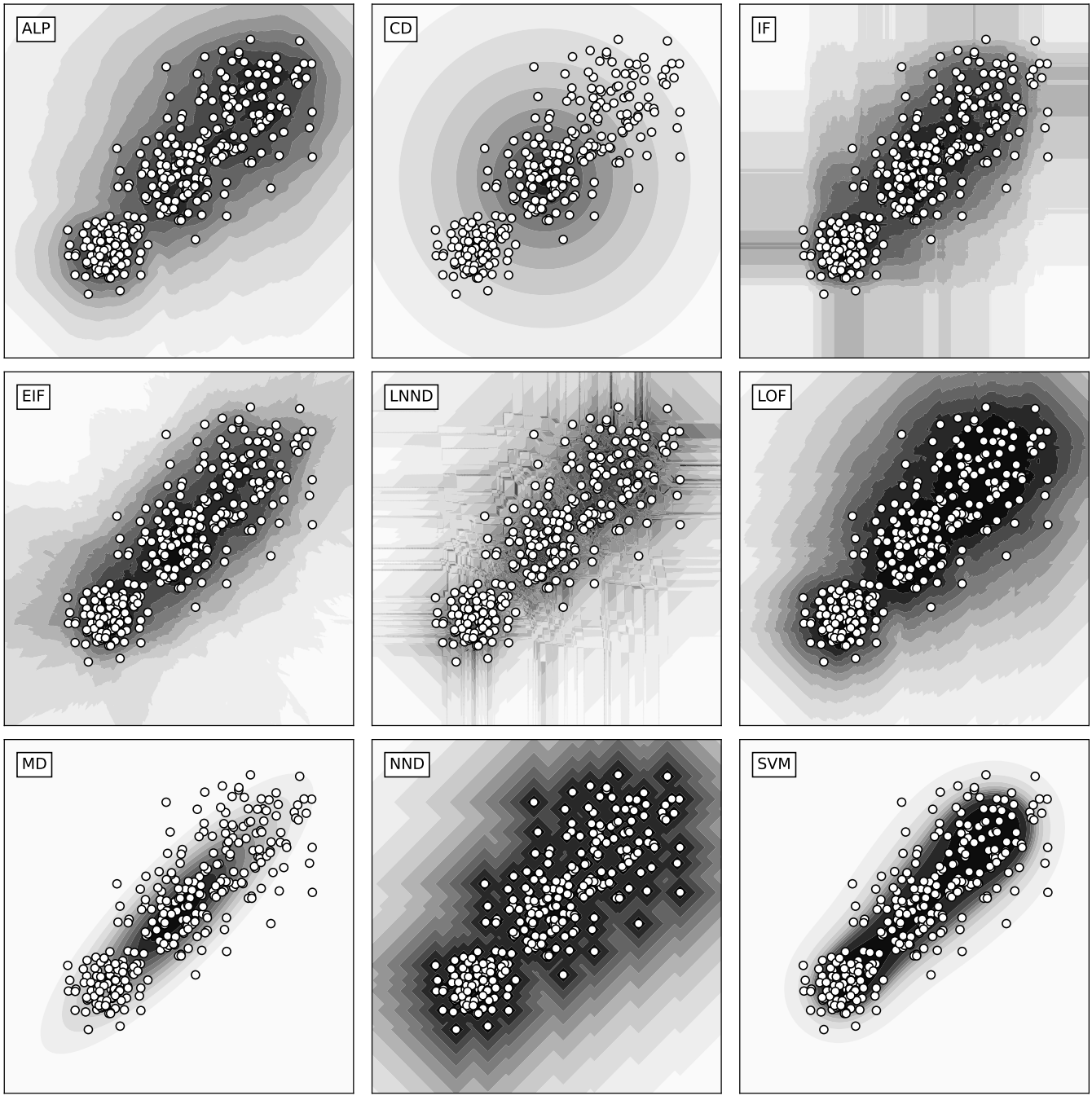

Fig. 2. Contour lines of data descriptor models in fuzzy-rough-learn, constructed on a randomly generated toy dataset.

d_i of the l nearest neighbours $\mathrm{NN}_j(y)$ of y, with linearly decreasing weights w^l:

$$D_i(y) := \sum_{j \leq l} w_j^l \cdot d_i(\mathrm{NN}_j(y)). \tag{4}$$

We use this to obtain the ith localised proximity of y:

$$\mathrm{lp}_i(y) = \frac{D_i(y)}{D_i(y) + d_i(y)}, \tag{5}$$

$$\tag{6}$$

and define the average localised proximity of y as the soft maximum of its k localised proximities, induced by k linear decreasing weights:

$$\operatorname*{soft\,max}_{i \leq k} \mathrm{lp}_i(y). \tag{7}$$

2) CD: Centre Distance, a very simple data descriptor based on the distance to the origin (vector size). This can be given meaning by choosing a suitable preprocessor which centres the data on the origin. The default preprocessor centres the mean/centroid, and divides each attribute by its standard deviation.

3) EIF: Extended Isolation Forest [10]. Generalised variant of IF, that allows tree splits along hyperplanes that are not necessarily parallel to the axes. Wrapper for the implementation in the eif package.[3]

4) IF: Isolation Forest [11]. Constructs trees with random splits on the target data, and uses the path length through these trees as a measure for the similarity of an instance to the target data. Wrapper for the implementation in scikit-learn.

5) LNND: Localised Nearest Neighbour Distance [12], [13]. Based on the ratio between the distance of a test instance to its kth nearest neighbour x, and the distance between x and its own kth nearest neighbour.

6) LOF: Local Outlier Factor [14]. Based on the concept of the ith reachability distance $\mathrm{rd}_i(x)$ of a point x, which is the maximum of the distance between x and its ith nearest neighbour in the target data, and the kth nearest neighbour distance of the latter. Then the inverse of the mean reachability distances of x is its local reachability density $\mathrm{lrd}(x)$:

$$\mathrm{lrd}(x) := \frac{1}{\frac{1}{k}\sum_{i \leq k} \mathrm{rd}_i(x)}. \tag{8}$$

Finally, the local outlier factor of a test instance y is obtained by dividing the mean local reachability density of its nearest neighbours $\mathrm{NN}_j(y)$ by its own local reachability density:

$$\frac{\frac{1}{k}\sum_{j \leq k} \mathrm{lrd}(\mathrm{NN}_j(y))}{\mathrm{lrd}(y)}. \tag{9}$$

[3]https://github.com/sahandha/eif

TABLE II
CONVENIENCE FUNCTIONS FOR LINEAR NORMALISERS IN FUZZY-ROUGH-LEARN.

	Centre	Dispersion
`IQRNormaliser`	`midhinge`	`interquartile_range`
`MaxAbsNormaliser`		`maximum_absolute_value`
`RangeNormaliser`	`midrange`	`total_range`
`Standardiser`	`mean`	`standard_deviation`

7) MD: Mahalanobis Distance [15]. Generalisation of the number of standard deviations that a point may be removed from the mean in one dimension, to multinomial distributions. For a test instance y, this is defined as

$$\sqrt{(y-\mu)^T S^{-1} (y-\mu)}, \tag{10}$$

where μ is the mean of the target data, and S its covariance matrix. Square Mahalanobis distance follows a χ^2-distribution with m degrees of freedom, and we obtain a p-value in $[0, 1]$ by applying the relevant cumulative distribution function and subtracting the result from 1, which expresses the probability of drawing y if the target data were drawn from a Gaussian distribution.

8) NND: Nearest Neighbour Distance [16]. Based on the distance between a test instance y and its kth nearest neighbour in the target data. Our implementation is more general, allowing for the aggregation of the k nearest neighbour distances with a weight vector. Internally, this is used to calculate upper and lower approximation memberships in FRNN classification.

9) SVM: Support Vector Machine [17], [18]. Fits a hyperplane between the target data and the origin, and generates predictions on the basis of the distance to this hyperplane. Wrapper for the implementation in scikit-learn.

B. Feature preprocessors

1) LinearNormaliser: Unsupervised. Rescales the data by centring it on the specified measure of central tendency and/or dividing by the specified measure of dispersion. We provide a number of convenience functions (Table II), but the more general class allows users to define their own variants.

2) SAE: Shrink Autoencoder [4]. Unsupervised, designed to make target data easier to separate from other data by data descriptors. Learns a $\lfloor\sqrt{m}\rfloor + 1$-dimensional latent representation of the target data, which is induced to shrink around the origin by the cost function, which balances reconstruction error against the Euclidean norm of the latent representation.

This is a reimplementation using the Keras and TensorFlow framework, based on the code provided by the original authors.[4]

The SAE neural network consists of six dense layers. The first three layers encode the data by linearly decreasing the number of features from m to $\lfloor\sqrt{m}\rfloor + 1$, while the last three layers use the same (tied) weights to decode the data again.

[4]https://github.com/vanloicao/SAEDVAE

We use the hyperbolic tangent activation function and Glorot uniform weight initialisation [19].

The target data is split into a training set (80%) and a validation set (20%). Batch size is equal to 5% of the training set, with a maximum of 100. Validation accuracy is calculated every 5 epochs. The network is trained for 1000 epochs, or until the early stopping criterion is satisfied, which is the case when validation accuracy has not substantially increased for a number of epochs corresponding to 400 batches. The network is trained with the ADADELTA optimiser [20] with an initial learning rate of 0.01.

3) VectorSizeNormaliser: Unsupervised. Projects all vectors onto the unit sphere. Typically used in natural language processing (NLP) for frequency counts, when only relative frequencies are deemed important. What is commonly called the cosine dissimilarity can then be obtained by measuring the squared Euclidean distance.

C. Regressors

1) FRNN: Fuzzy Rough Nearest Neighbour regression [5]. Let X be the training set and $d : X \longrightarrow \mathbb{R}$ the target attribute. For a test instance y, the fuzzy tolerance class $R_d(\cdot, \mathrm{NN}_i(y))$ of its ith nearest neighbour $\mathrm{NN}_i(y)$ is the fuzzy set defined as:

$$R_d(\cdot, \mathrm{NN}_i(y))(x) := \frac{|d(\mathrm{NN}_i(y)) - d(x)|}{d_{\max} - d_{\min}}, \tag{11}$$

for any x in X. We use this to define a weight w_i, equal to the mean membership degrees of y in the (strict) upper and lower approximations of $R_d(\cdot, \mathrm{NN}_i(y))$:

$$w_i := (\overline{R_d(\cdot, \mathrm{NN}_i(y))}(y) + \underline{R_d(\cdot, \mathrm{NN}_i(y))}(y))/2. \tag{12}$$

The upper approximation membership is equal to at least the similarity between y and $\mathrm{NN}_i(y)$, but can be higher if there is a closer neighbour with a sufficiently similar target value. The lower approximation membership is equal to at least the distance to the closest neighbour $\mathrm{NN}_1(y)$, but can be higher if the target value of $\mathrm{NN}_1(y)$ is sufficiently similar to that of $\mathrm{NN}_i(y)$. In sum, greater weight is given to the values of closer neighbours, and to values of neighbours that reinforce each other by having similar target values.

Finally, these weights are used to predict a target value for y as the weighted sum of the target values of its k neighbours (default 20):

$$d(y) := \sum_{i \leq k} w_i \cdot d(\mathrm{NN}_i(y)) / \sum_{i \leq k} w_i. \tag{13}$$

While the calculation of the upper approximation membership values can be restricted to the k nearest neighbours of y, the calculation of the lower approximation membership values in principle requires considering all instances in the training data. Therefore, FRNN regression does not at present scale well to large datasets.

VI. Utilities

In this section, we discuss all the other functions included in fuzzy-rough-learn. These can be used as input for various hyperparameters of the algorithms discussed in the previous section.

A. Array functions

A collection of diverse utility functions for working with NumPy arrays. `first`, `last`, `least` and `greatest` return, respectively, the k first, last, least and greatest elements along the specified axis of an array. These functions are used internally by the functions `soft_head`, `soft_tail`, `soft_min`, `soft_max`, which additionally take a weight function (Subsection VI-I), and use this to aggregate the k values along the specified axis. `soft_max` and `soft_min` are implementations of the soft maximum and minimum from Section II.

`remove_diagonal` takes an n by n square matrix, and removes the diagonal to obtain an n by $n-1$ matrix. This can be used to remove comparisons of an instance with itself. `div_or` is simple division of an array, but wherever the division results in `NaN` (e.g. from $0/0$), a fallback value (default 1) is substituted.

B. Dispersion measures

Functions that take a dataset, and return a one-dimensional array, indicating the dispersion of the dataset in each dimension. `total_range` is the difference between the largest and smallest values. `interquartile_range` ignores extreme values, and returns the difference between the third and first quartile. `maximum_absolute_value` is the maximum of the absolute values of the maximum and the minimum. `standard_deviation` is the Euclidean distance between the dataset and its mean.

C. Location measures

Functions that take a dataset, and return a one-dimensional array, indicating the relevant location of the dataset in each dimension. `minimum`, `maximum`, `mean` and `median` are thin wrappers for the corresponding NumPy functions, with the only specific difference that `NaN` values are ignored. `midrange` is the mean of the minimum and the maximum, while `midhinge` is the mean of the first and third quartiles.

D. Neighbour search methods

Algorithms that construct a model which returns, for some $0 < k \leq n$, the k nearest neighbours of each query instance and their distances. The two algorithms `BallTree` [21] and `KDTree` [22] are wrappers for the corresponding implementations in `scikit-learn`. By subclassing the abstract base class, users can substitute their own nearest neighbour search algorithms, such as Hierarchical Navigable Small World (HNSW) [23], [24], which scales well to large datasets, but delivers only approximately accurate results.

E. Parametrisations

Utility functions that can be used to express a dependency of a value on another value. The concrete use case in this library is to let the number of nearest neighbours k depend on the number of available instances n. `multiple` multiplies n with the provided coefficient (typically a fraction). `log_multiple` multiplies $\log n$ with the provided coefficient.

F. T-norms

Triangular norms, commutative, associative, non-decreasing binary operators on $[0, 1]$ with identity 1, expressed here as aggregation functions that take an array and reduce it along the indicated axis. The `goguen_t_norm` is the product. The `heyting_t_norm`, also known as the Gödel norm, is the minimum. Finally, the `lukasiewicz_t_norm` is equal to the sum of values, minus their count reduced by one.

G. Transformations

Functions to transform distance values in $[0, \infty]$ or signed distance values in $[-\infty, \infty]$ into similarity values in $[0, 1]$. `shifted_reciprocal` transforms distance values with $x \longmapsto \frac{1}{1+x}$, while `truncated_complement` uses $x \longmapsto \max(0, 1 - x)$. `contract` transforms signed distance values with $x \longmapsto \frac{x}{2*(|x|+c)} + 0.5$.

H. Vector size measures

Functions from real vector spaces to $[0, \infty]$ like norms. Can also be used as dissimilarity measures through application to $y - x$.

At present contains a single parametrisable function, `MinkowskiSize`, which encompasses a family of functions. For $p \in (0, \infty)$, it assigns to an m-dimensional vector v the size $(\sum |v_i|^p)^{\frac{1}{p}}$. The special case $p = 1$ corresponds to the Boscovich norm, also known as city block, Manhattan, rectilinear or taxicab norm. The special case $p = 2$ corresponds to the Euclidean norm, also known as Pythagorean norm.

If the additional boolean parameter `unrooted` is set to true, exponentiation by $\frac{1}{p}$ is omitted. This lets us obtain the squared Euclidean norm and generalisations thereof.

The special cases $p = 0$ and $p = \infty$ are defined through their limits. For $p = 0$, the rooted variant evaluates to ∞ if v has more than one non-zero coefficient, to 0 if all its coefficients are zero, and to the only non-zero coefficient otherwise, while the unrooted variant is equal to the number of non-zero coefficients of v. The latter is commonly known as the Hamming size.

For $p = \infty$, the rooted variant is the maximum of all coefficients. This is the Chebyshev norm, also known as chessboard or maximum norm. The unrooted variant evaluates to ∞ if v has at least one coefficient larger than 1, and to the number of coefficients equal to 1 otherwise.

Finally, if the boolean parameter `scale_by_dimensionality` is set to true, the result is scaled linearly to ensure that all vectors in $[0, 1]^m$ have size in $[0, 1]$.

TABLE III
WEIGHT FUNCTIONS IN FUZZY-ROUGH-LEARN. q IS A NON-DECREASING, REGULAR QUANTIFIER $[0, 1] \longrightarrow [0, 1]$.

`ConstantWeights`	$\frac{1}{k}, \frac{1}{k}, \ldots, \frac{1}{k}$
`ExponentialWeights`	$\frac{2^{k-1}}{2^k-1}, \frac{2^{k-2}}{2^k-1}, \ldots, \frac{2^0}{2^k-1}$
`LinearWeights`	$\frac{k}{k(k+1)/2}, \frac{k-1}{k(k+1)/2}, \ldots, \frac{1}{k(k+1)/2}$
`QuantifierWeights` [6]	$\frac{q(\frac{1}{k})}{q(\frac{0}{k})}, \frac{q(\frac{2}{k})}{q(\frac{1}{k})}, \ldots, \frac{q(\frac{k}{k})}{q(\frac{k-1}{k})}$
`ReciprocallyLinearWeights`	$\frac{1}{1\cdot\sum_{i\leq k}\frac{1}{i}}, \frac{1}{2\cdot\sum_{i\leq k}\frac{1}{i}}, \ldots, \frac{1}{k\cdot\sum_{i\leq k}\frac{1}{i}}$

I. Weights

Parametrisable functions that take a positive integer k and return a weight vector, listed in Table III. Linear and reciprocally linear weights are also known as *additive* and *inverse additive* in the literature. These weights can be used as input for the soft head, maximum, minimum and tail functions (Subsection VI-A).

J. Other: postprocessing functions

In addition to the utilities listed above, we also provide some functions to help with postprocessing classifier predictions.

`select_class` takes a two-dimensional array of records and class predictions, and returns a one-dimensional array that contains for each record the class with the highest score. In addition, an abstention threshold can be set, such that records with no class score above this threshold are labelled separately.

`discretise` takes an array of label scores and discretises these to either 0 or 1 depending on a threshold value. This can be used both for the scores produced by data descriptors and for multilabel classifiers.

`probabilities_from_scores` rescales class scores such that they sum to 1 for each record.

VII. FUTURE WORK

Like many software libraries, fuzzy-rough-learn is a permanent work-in-progress. Since it is still in its early stages of development, its shape has not crystallised into a definite form, and there are a number of unresolved issues. We are still looking for a way to better handle components, like the SAE preprocessor, that rely on optional dependencies. In addition, we are looking for a way to show how algorithms from other libraries, like approximative nearest neighbour search algorithms, can be used together with the algorithms in fuzzy-rough-learn, without turning those libraries into dependencies.

Another long-term goal is the inclusion of low-level algorithms, for which we currently use scikit-learn as a backend. Apart from making fuzzy-rough-learn more self-contained, this would also make it easier to expand our algorithms with new functionality.

In the near term, we hope to make pre- and post-processing more convenient to the user. In particular, we want to make it easy to use datasets with categorical and missing values, to perform hyperparameter optimisation, and to interpret classifier scores heuristically on the basis of training data.

ACKNOWLEDGEMENT

The research reported in this paper was conducted with the financial support of the Odysseus programme of the Research Foundation – Flanders (FWO).

REFERENCES

[1] O. U. Lenz, D. Peralta, and C. Cornelis, "fuzzy-rough-learn 0.1: a Python library for machine learning with fuzzy rough sets," in *IJCRS 2020: Proceedings of the International Joint Conference on Rough Sets*, ser. Lecture Notes in Artificial Intelligence, vol. 12179. Springer, 2020, pp. 491–499.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in Python," *J Mach Learn Res*, vol. 12, no. 85, pp. 2825–2830, 2011.

[3] D. M. J. Tax, "One-class classification: Concept learning in the absence of counter-examples." Ph.D. dissertation, Technische Universiteit Delft, 2001.

[4] V. L. Cao, M. Nicolau, and J. McDermott, "Learning neural representations for network anomaly detection," *IEEE Trans Cybern*, vol. 49, no. 8, pp. 3074–3087, 2019.

[5] R. Jensen and C. Cornelis, "Fuzzy-rough nearest neighbour classification and prediction," *Theor Comput Sci*, vol. 412, no. 42, pp. 5871–5884, 2011.

[6] R. R. Yager, "On ordered weighted averaging aggregation operators in multicriteria decisionmaking," *IEEE Trans Syst Man Cybern*, vol. 18, no. 1, pp. 183–190, 1988.

[7] J. H. Saltzer, "The origin of the "MIT license"," *IEEE Ann Hist Comput*, vol. 42, no. 4, pp. 94–98, 2020.

[8] S. Vluymans, A. Fernández, Y. Saeys, C. Cornelis, and F. Herrera, "Dynamic affinity-based classification of multi-class imbalanced data with one-versus-one decomposition: a fuzzy rough set approach," *Knowl Inf Syst*, vol. 56, no. 1, pp. 55–84, 2018.

[9] O. U. Lenz, D. Peralta, and C. Cornelis, "Average Localised Proximity: A new data descriptor with good default one-class classification performance," *Pattern Recognit*, vol. 118, p. 107991, 2021.

[10] S. Hariri, M. Carrasco Kind, and R. J. Brunner, "Extended Isolation Forest," *IEEE Trans Knowl Data Eng*, vol. 33, no. 4, pp. 1479–1489, 2021.

[11] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *ICDM 2008: Proceedings of the Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422.

[12] D. de Ridder, D. M. J. Tax, and R. P. W. Duin, "An experimental comparison of one-class classification methods," in *ASCI'98: Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*. ASCI, 1998, pp. 213–218.

[13] D. M. J. Tax and R. P. W. Duin, "Outlier detection using classifier instability," in *SSPR/SPR 1998: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition and Structural and Syntactic Pattern Recognition*, ser. Lecture Notes in Computer Science, vol. 1451. Springer, 1998, pp. 593–601.

[14] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," in *SIGMOD 2000: Proceedings of the ACM International Conference on Management of Data*, vol. 29, no. 2. ACM, 2000, pp. 93–104.

[15] P. C. Mahalanobis, "On the generalized distance in statistics," *Proc Natl Inst Sci India*, vol. 2, no. 1, pp. 49–55, 1936.

[16] E. M. Knorr and R. T. Ng, "A unified notion of outliers: Properties and computation." in *KDD-97: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*. AAAI, 1997, pp. 219–222.

[17] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," Microsoft Research, Redmond, Washington, Tech. Rep. MSR-TR-99-87, 1999.

[18] ——, "Estimating the support of a high-dimensional distribution," *Neural Comput*, vol. 13, no. 7, pp. 1443–1471, 2001.

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *AISTATS 2010: Proceedings of the thirteenth international conference on artificial intelligence and statistics*, ser. Proceedings of Machine Learning Research, vol. 9. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[20] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," 2012. [Online]. Available: http://arxiv.org/abs/1212.5701

[21] S. M. Omohundro, "Five balltree construction algorithms," International Computer Science Institute, Berkeley, California, Tech. Rep. TR-89-063, 1989.

[22] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[23] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans Pattern Anal Mach Intell*, vol. 42, no. 4, pp. 824–836, 2020.

[24] O. U. Lenz, D. Peralta, and C. Cornelis, "Scalable approximate FRNN-OWA classification," *IEEE Trans Fuzzy Syst*, vol. 28, no. 5, pp. 929–938, 2020.