Centroid.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: 18 January 2018

The use of this program requires the following packages: numpy, copy, random, matplotlib, scipy, colorsys. Note that within this python file are many functions that go un-explained in this documentation because many of them are back-end functions that the user need not to generally concern himself with. However, we will explain a few functions that may be beneficial for the user to be aware of - but first will will start with the Centroid class.

class Centroid.**Centroid**(*`method`=‘smallestdisk’, `weights`=‘ones’, `norm`=True, `predictor`=‘radius’*)

A cluster interpretation tool. This tool is limited in dimensionality depending on the parameters chosen. For example, using ‘poly’ as the centroid detection method or ‘projection’ as the predictor calls the scipy.spatial.ConvexHull package whose accuracy is known to be limited to 9 dimensions.

We coded this tool and structured the documentation similar to the style of sci-kit learn.

Parameters:

`method` : string, optional (default=‘smallestdisk’)

Specifies how the center of a cloud of points should be calculated. The options are:

- ‘smallestdisk’ - Determines the smallest n-dimensional sphere which encompasses all of the points using Fischer’s fast smallest enclosing ball algorithm [1]. We add a slight modification to the initial state of the algorithm- rather than initializing the center at a random point, we initialize on the best representative point (explained in ‘bestrep’ option).
- ‘boundingbox’ - Determines the smallest n-dimensional box which encompasses all of the points.
- ‘median’ - The median of each dimension is calculated.
- ‘bestrep’ - The point whose maximum distance to all other points is a minimum.
- ‘poly’ - Finding the center of a convex hull using a slightly modified version of the algorithm Maire describes [2].
- ‘average’ - The mean of each dimension is calculated.
- ‘push&pull’ - Attempts to maximize inter cluster distances and minimize intra cluster distances using spring like pulling forces (Hooke’s law) and electromagnetic pushing force (Coulomb’s Law) inspired by Fruchterman-Reingold’s force directed graph drawing algorithm [3].

If given an input other than one of the above then no center calculation will take place during the fitting step.

`weights` : string or list, optional (default=‘ones’)

The weights for each feature. The only string option available is ‘ones’ which will automatically give a weight of one to each feature. Otherwise the input is a list of numerical values corresponding to the column weights in order.

`norm` : bool or string, optional (default=True)

Normalizes the columns of the input data to the range [0,1] if True. It also accepts ‘neg’ as an input which normalizes all negative values of the features to the range [-0.5,0] and all the positive values to [0,0.5]. Thus maintaining a total range length of 1. Note that if ‘neg’ is specified then a feature is only transformed to the space [-0.5,0.5] if the minimum value of the feature is negative, otherwise it is normalized to [0,1].

`predictor` : string, optional (default='radius')

Prediction method on new samples, specifically for the `predict` and `predict_proba` methods. Takes as inputs:

- 'nearest' - Calculates the distance between the new sample s and each center c_i: $||s - c_i||$ for each cloud i.
- 'radius' - This corresponds to the "radial normalization" described in our paper. Calculates the distance between the new sample and each center normalized by its radius r_i (determined as the distance to the furthest point from the center): $\frac{||s-c_i||}{r_i}$ for each cloud i.
- 'closestborder' - Each feature is given its own radius which is calculates as the furthest point in the feature from the center. Then the distance for new samples are calculated as the maximum normalized feature distance.
- 'elliptical' - Each feature is given its own radius. Then calculation of distance of the new sample is similar to 'radius' but now we consider elliptical cluster shape to be possible. The mathematics is not described here because experimental results showed that prediction was not as powerful as the radius method (which is also a much simpler method in regards to prediction).
- 'projection' - This corresponds to the "projected hyperplane normalization" described in our paper. Calculates the distance between the new sample and each center normalized by its nearest projected hyperplane in the convex hull (determined by utilizing the package scipy.spatial.**ConvexHull**). More specifically, assuming that each hyperplane takes the form:

$$k_0x_0 + k_1x_1 + \cdots + k_nx_n + K = 0 \tag{1}$$

Then let the projected vector from the center to the new sample be given by and its direction be given by:

$$\vec{v} = s - c_j$$

$$\vec{d} = \frac{\vec{v}}{||\vec{v}||}$$

Where $\vec{d}$ must necessarily be a vector of n elements, where the j^{th} element is denoted as $\vec{d_j}$ and the j^{th} element of center i is denoted $c_{i,j}$. Then a parameterization of the line, with the bounds $-\infty < t < \infty$, takes the form:

$$(x_0, x_1, \ldots, x_n) = (c_{i,0} + \vec{d_0}t, c_{i,1} + \vec{d_1}t, \ldots, c_{i,n} + \vec{d_n}t) \tag{2}$$

Then by substitution of Eq 2 into Eq 1 we have:

$$t = \frac{-K - k_0c_{i,0} - k_1c_{i,1} - \cdots - k_nc_{i,n}}{k_0\vec{d_0} + k_1\vec{d_1} + \cdots + k_n\vec{d_n}} \tag{3}$$

For every hyperplane h in the convex hull of cloud i, substituting the result of Eq 3 into Eq 2, gives us the point of intersection denoted as y_h. We choose to use the hyperplane whose projected distance is a minimum (i.e. the distance from the center to the intersection):

$$pd_i = \min_h distance(c_i, y_h) \tag{4}$$

The distance function is conditional- coded as follows:

$$distance(c_i, y_h) = \begin{cases} \infty, & \text{condition 1} \\ \text{ERROR}, & \text{condition 2} \\ \infty, & \text{condition 3} \\ ||c_i - y_h||, & otherwise \end{cases}$$

condition 1: The divisor of Eq 3 is equal to 0. This means that the hyperplane and vector are parallel to each other, hence they will never intersect. Thus the distance is considered to be infinite.

condition 2: The numerator of Eq 3 is greater than or equal to 0. This would mean the center is located either on the boundary or outside the hyperplane- hence we raise an **AssertionError** because poor calculation of the center: this is known to sometimes occur when the methods 'median' or 'bestrep' are used, and is almost guaranteed to occur when 'push&pull' is used.

condition 3: $t < 0$. This would assume that the intersection occurs in the opposite direction of $\vec{d}$, hence the distance is changed to ∞.

Thus the projected hyperplane normalization is calculated as $\frac{||s-c_i||}{pd_i}$ for each cloud i.

Attributes:

`method_` : string

The method used to calculate the centers.

`C_` : numpy array of shape = (n_classes,)

Holds the unique set of class names in a numpy array after `fit` is performed.

`weights_` : list (or numpy array)

List of weights for each feature. This is defaulted to a list unless if a numpy array is inputted as one of the parameters.

`norm_` : bool or string

Returns the norm parameter that was inputted.

`predictor_` : string

Returns the predictor parameter that was inputted.

`LT_` : dictionary or numpy array of shape = (2,n_features)

Returns a dictionary if 'neg' was inputted under the `norm` parameter, otherwise gives a 2 row by n_feature column numpy array, where the first row corresponds to the slope (m) and the second row corresponds to the y-intercept (b). If dictionary is the output, then each key corresponds to the feature number, and within the value is another dictionary of positive ('p') and/or negative results ('n') each with a list where the first value is the slope and the second is the y-intercept. The values are stored in this attribute after `fit` is performed.

`D_` : dictionary

After `fit` is called, this attribute stores as its keys each unique class with a numpy array as its value of shape (k_samples,n_features) corresponding to data X inputted into `fit` which is unique to a class. Thus the number of rows for each class is not necessarily the same.

`centers_` : dictionary

Each key corresponds to a unique class holding as its value a numpy array of shape (n_features,) which corresponds to the center calculated for that class- after `fit` is performed

`radii_` : dictionary

Each key corresponds to a unique class with a value corresponding to the radius calculated after `fit` is performed. If the 'radius' is inputted as the `predictor`, then this attribute holds a radius for every unique class. Otherwise, this attribute will only hold a value for each class if 'smallestdisk' or 'bestrep' are used as the `method` because the calculation of the radius is inherent in the centroid detection method. If 'elliptcal' or 'closestborder' is chosen as the `predictor` then the dictionary will hold as its values a numpy array of radii (with its length corresponding to the number of features) - as the radii of each feature space is different in regard to these methods.

`hulls_` : dictionary

Each key corresponds to a unique class with the convex hull being stored as the value- only after `fit` is performed with 'projection' being used as the `predictor`. If 'projection' is not specified, then this attribute will only hold values for each class if 'poly' is used as the `method` because calculating the convex hull is inherent in the method. The convex hull is calculated using the scipy.spatial.**ConvexHull** package.

Methods:

Summary:

`LT (x[, DC])`	Linearly transforms `x` based on the `LT_` found from `fit`
`inv_LT (x[, DC])`	Inversely transforms `x` based on the `LT_` found from `fit`
`get_inv_centers ()`	Yields the center information in the unnormalized region
`fit (X,y)`	Builds the Centroid Interpretation based on training set (`X`, `y`)
`predict (X)`	Predict class(es) of `X` depending on on the predictor_
`predict_proba (X)`	Predict class probability(ies) of `X` depending on predictor_
`plot ([...])`	Plots the centroid rules on a number line - only after `fit` is run.

Explanation:

`LT` (*x, DC=False*)

Linearly transforms `x` based on the `LT_` found from `fit`.

Parameters:

`x` : vector (numpy array), shape = [n_features,]

The vector to be linearly transformed based on LT_ results.

`DC` : boolean, (default=False)

Stands for Deep Copy. If DC is left False then the original `x` will be changed to be identical to the output.

Returns:

`x` : vector (numpy array), shape = [n_features,]

The transformed vector.

`inv_LT` (*x, DC=False*)

Inversely transforms `x` based on the `LT_` found from `fit`.

Parameters:

`x` : vector (numpy array), shape = [n_features,]

The vector to be inversely transformed based on LT_ results.

`DC` : boolean, (default=False)

Stands for Deep Copy. If DC is left False then the original `x` will be changed to be identical to the output.

Returns:

`x` : vector (numpy array), shape = [n_features,]

The inversely transformed vector.

`get_inv_centers` ()

Yields the center information in the unnormalized region.

Returns:

`iC` : dictionary

For each center (class) in `centers_` the inverse linear transformed is applied (`inv_LT`) to attain the center information in the unnormalized space. Eeach key in `iC` is identical to the keys in `centers_`.

`fit` (*X, y*)

Builds the Centroid Interpretation based on training set (`X`, `y`). The centroids will be approximated in accordance to `method_`.

Parameters:

`X` : array-like (numpy array), shape = [n_samples, n_features]

The training input samples.

`y` : array-like (numpy array), shape = [n_samples,]

The target values (class labels) as integers or strings.

Returns:

`self` : object

Returns self.

`predict` (*X*)

Predict class(es) of `X` depending on on the predictor_ and in accordance to the names provided in the target (`y`) during `fit`.

Parameters:

`X` : array-like (numpy array), shape = [n_samples, n_features]

The input samples to be predicted.

Returns:

`y` : string/int or array of shape = [n_samples,] The predicted class(es). If `X` consists only of one sample then it returns a single string/int (depending on class labels from `y` in `fit`), otherwise returns a numpy array.

`predict_proba` (*X*)

Predict class probability(ies) of `X` depending on predictor_.

Parameters:

`X` : array-like (numpy array), shape = [n_samples, n_features]

The input samples to be predicted.

Returns:

`p` : array of shape = [n_samples, n_classes]

Probability score of each class in the order corresponding to the classes in `C_`. Warning: class membership is chosen (prediction) based on the class with the minimum distance (after normalization-if specified), which do not have inherent probability values. Thus we do not recommend the use of `predict_proba` because the results will not necessarily be coherent. That being said, probabilities are calculated as follows: Let S be the sum of the distances. Let wS be the sum of the inverted distances (e.g. let c_1 be the distance to class one, then the inverted distance is $\frac{S}{c_1}$). Then for all classes i, $P(c_i) = \frac{S}{c_i * wS}$. Note that this fails when there exists j such that $c_j = 0$. For this case we return an array where the probability of all classes are zero except for the class j- which is given a probability of 1 (100%). This assumes that there can only be one class for where the distance to the center is zero. If there are more than one, then it will only pick the first encounter.

`plot` (*feat_names=None, class_colors={}, title=''Centroid Rules'', c2f=0.05, f2b=0.01, b2b=0.15, b2c=0.05, b_size=0.05, b2s_ratio=0.9, b2u=0.005, c_size=18, f_size=10, u_size=10, s_size=12*)

Plots the centroid rules on a number line - only after `fit` is run.

Parameters:

`feat_names` : None or array-like (numpy array or list), shape = [n_features,], (default=None)

The names of the features corresponding to the column index value from the array `fit` was trained on.

`class_colors` : dictionary, (default={})

The keys of the dictionary should be the names of the class, and the value should be an rgb code or a color-string code recognized by `matplotlib`.

`c2f` : float, (default=0.05)

Spacing between the class name and the feature bar (alternatively it is the spacing from the class name to the feature name for the instance when the feature names are printed).

`f2b` : float, (default=0.01)

Spacing between the feature name and the feature bar (only for the first instance when the feature names are printed).

`b2b` : float, (default=0.15)

Spacing between feature bars in the vertical direction (this parameter is currently unused for this function).

`b2c` : float, (default=0.05)

Spacing between the feature bar and the next class name.

`b_size` : float, (default=0.05)

Height of the feature bar.

`b2s_ratio` : float, (default=0.9)

Ratio of the space controlling the feature bar to the space between bars in the horizontal direction. For example, if b2s_ratio is set to 1 then there will be no space between bars in the horizontal direction.

`b2u` : float, (default=0.005)

Spacing between the feature bar and its units (i.e. the centroid).

`c_size` : integer or float, (default=18)

Fontsize of the class names and the title.

`f_size` : integer or float, (default=10)

Fontsize of the feature names.

`u_size` : integer or float, (default=10)

Fontsize of the units (i.e. the centroid).

`s_size` : integer or float, (default=12)

Fontsize of the description in margins (currently this parameter goes unused).

Backend Methods Summary (No Elaboration Given):

`normer (v,i,n,a)`	Normalizes feature to a range between [0,1] or [-0.5,0.5]
`euc (x1,x2)`	Calculates the 2-norm between x1 and x2
`eucM (x,M[,keep_track,use_max])`	Calculates the 2-norm between x and each M
`NP ([Cs])`	Approximates center using the Polyhedral method
`PaP ([Cs])`	Approximates center using the Push and Pull method
`SD ([Cs])`	Approximates center using the Smallest Disk method
`BB ([Cs])`	Approximates center using the Bounding Box method
`BR ([Cs])`	Approximates center using the Best Representative method
`AVG ([Cs])`	Approximates center using the Mean (Average) method
`MED ([Cs])`	Approximates center using the Median method
`nearest (x,xc,c)`	Predicts membership via its nearest neighbor (un-normalized)
`bndradius (x,xc,c)`	Predicts membership via radial normalization
`projectedhyperplane (x,xc,c)`	Predicts membership via projected hyperplane normalization
`clasDis (x[,pred])`	Finds distance scores for prediction
`argmin (cD)`	Built-in argmin function to optimize speed

def Centroid.**warm_start_bubble**(*S*)

This function solves the smallest enclosing disk problem as Fischer et. al. have described it [1] with the exception of how they chose the initial starting point. We noticed that stochastically choosing an initial point led to varied results, hence we instead chose to use the initial point whose maximum distance to all the other points was a minimum, which consistently led to the most optimal answer. The parameter S is a numpy array (shape = [n_samples, n_features]) containing the points to enclose the disk around. The function returns two results: 1) the computed center in the form of a numpy array (shape = [n_features,]) and 2) the radius.

def Centroid.**comparative_plot**(*X*, *y*, *feat_names*=None, *class_colors*={}, *title*="Centroid Comparison", *c2f*=0.05, *f2b*=0.01, *b2b*=0.15, *b2c*=0.05, *b_size*=0.05, *b2s_ratio*=0.9, *b2u*=0.005, *bardiv*=30, *c_size*=18, *f_size*=10, *u_size*=10, *s_size*=12, *methods*=None, *save*='Fig6.pdf')

A plot is drawn that compares the centroids of each class using each method specified in the `methods` parameter. If `methods`=None then all the methods are used. `X` is the training data and `y` is the training labels that are needed for `fit`. `save` is the save name of the figure, switch `save`=False if you do not want to save the figure. `bardiv` controls the size of the rectangular representation of the centroid. The rest of the parameters are described in detail under the `plot` method of the Centroid class (because this is a sister function of that one).

References

[1] Fischer K, Gärtner B, Kutz M. Fast smallest-enclosing-ball computation in high dimensions. In: ESA. vol. 2832. Springer; 2003. p. 630–641.

[2] Maire F. An algorithm for the exact computation of the centroid of higher dimensional polyhedra and its application to kernel machines. In: Data Mining, 2003. ICDM 2003. Third IEEE International Conference on. IEEE; 2003. p. 605–608.

[3] Kobourov SG. Spring embedders and force directed graph drawing algorithms. arXiv preprint arXiv:12013011. 2012;.