pyDML Documentation

Release 0.0.1

Juan Luis Suárez Díaz

Jul 04, 2019

Current Algorithms:

The need of a similarity measure is very common in many machine learning algorithms, such as nearest neighbors classification. Usually, a standard distance, like the euclidean distance, is used to measure this similarity. The distance metric learning paradigm tries to learn an optimal distance from the data. This package provides the classic algorithms of supervised distance metric learning, together with some of the newest proposals.

CHAPTER 1

How to learn a distance?

There are two main ways to learn a distance in Distance Metric Learning:

- Learning a metric matrix M, that is, a positive semidefinite matrix. In this case, the distance is measured as

$$d(x, y) = \sqrt{(x - y)^T M (x - y)}.$$

- Learning a linear map L. This map is also represented by a matrix, not necessarily definite or squared. Here, the distance between two elements is the euclidean distance after applying the transformation.

Every linear map defines a single metric ($M = L^T L$), and two linear maps that define the same metric only differ in an isometry. So both approaches are equivalent.

1.1 Principal Component Analysis (PCA)

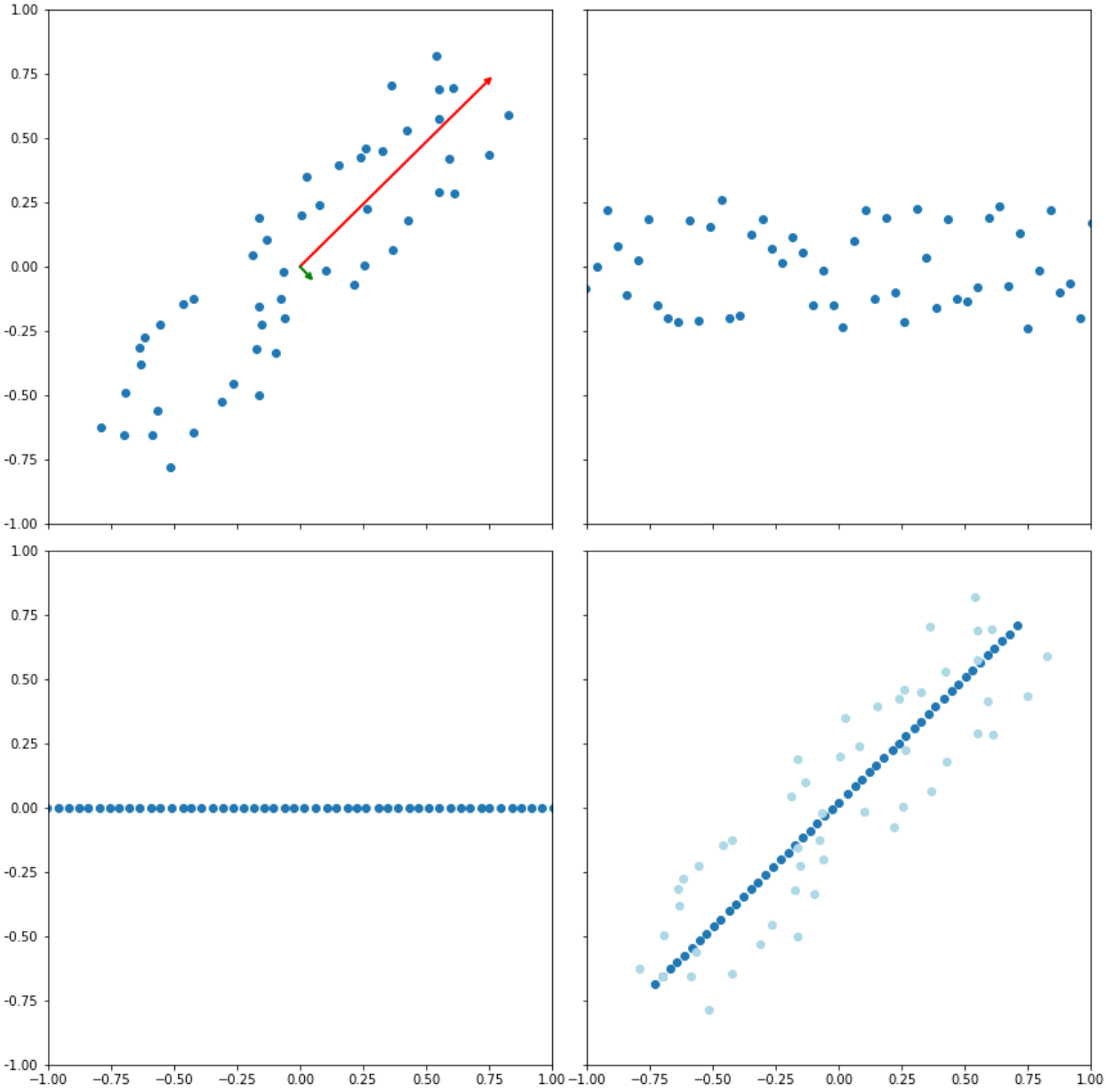
Principal Component Analysis is one of the most popular dimensionality reduction techniques. Note that this algorithm is not supervised, but it is still important as a preprocessing algorithm for many other supervised techniques.

PCA computes the first d' orthogonal directions for which the data variance is maximized, where d' is the desired dimensionality reduction.

The current PCA implementation is a wrapper for the Scikit-Learn PCA implementation.

Watch the full PCA documentation here.

1.1.1 Images



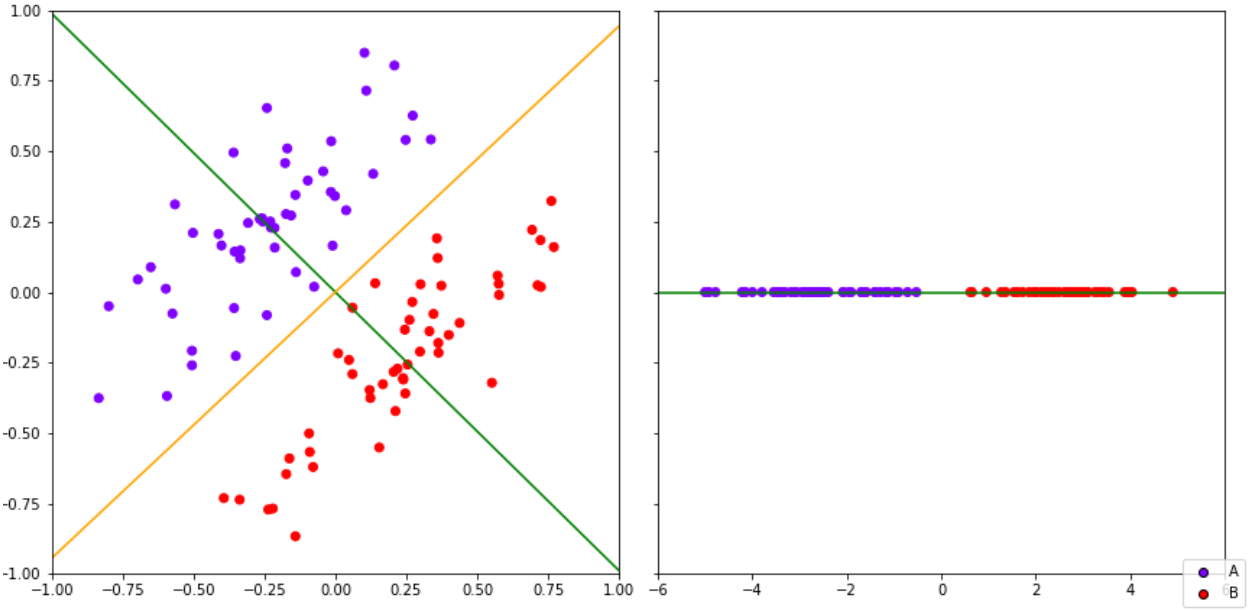
1.2 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis is a dimensionality reduction technique that finds the directions that maximize the ratio between the between-class variance and the within-class variances. This directions optimize the class separations in the projected space. The maximum number of directions this algorithm can learn is always lower than the number of classes.

The current LDA implementation is a wrapper from the Scikit-Learn LDA implementation.

Watch the full LDA documentation here.

1.2.1 Images



1.3 Average Neighborhood Margin Maximization (ANMM)

A dimensionality reduction technique that maximizes the sum of the average neighborhood margins for each point in the dataset. The average neighborhood margin for a point is calculated as the average of the difference between the sum of distances to the nearest neighbors with different class and the sum of distances to the nearest neighbors with same class.

Watch the full ANMM documentation here.

1.3.1 Images

1.3.2 References

Fei Wang and Changshui Zhang. “Feature extraction by maximizing the average neighborhood margin”. In: Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on. IEEE. 2007, pages 1-8.

1.4 Local Linear Discriminant Analysis (LLDA)

A local version of LDA. This algorithm performs with the same strategy as LDA, but it makes use of an affinity matrix, that is, a sparse matrix that measures the closeness between the samples, and that defines the local structure to be learned.

Watch the full LLDA documentation here.

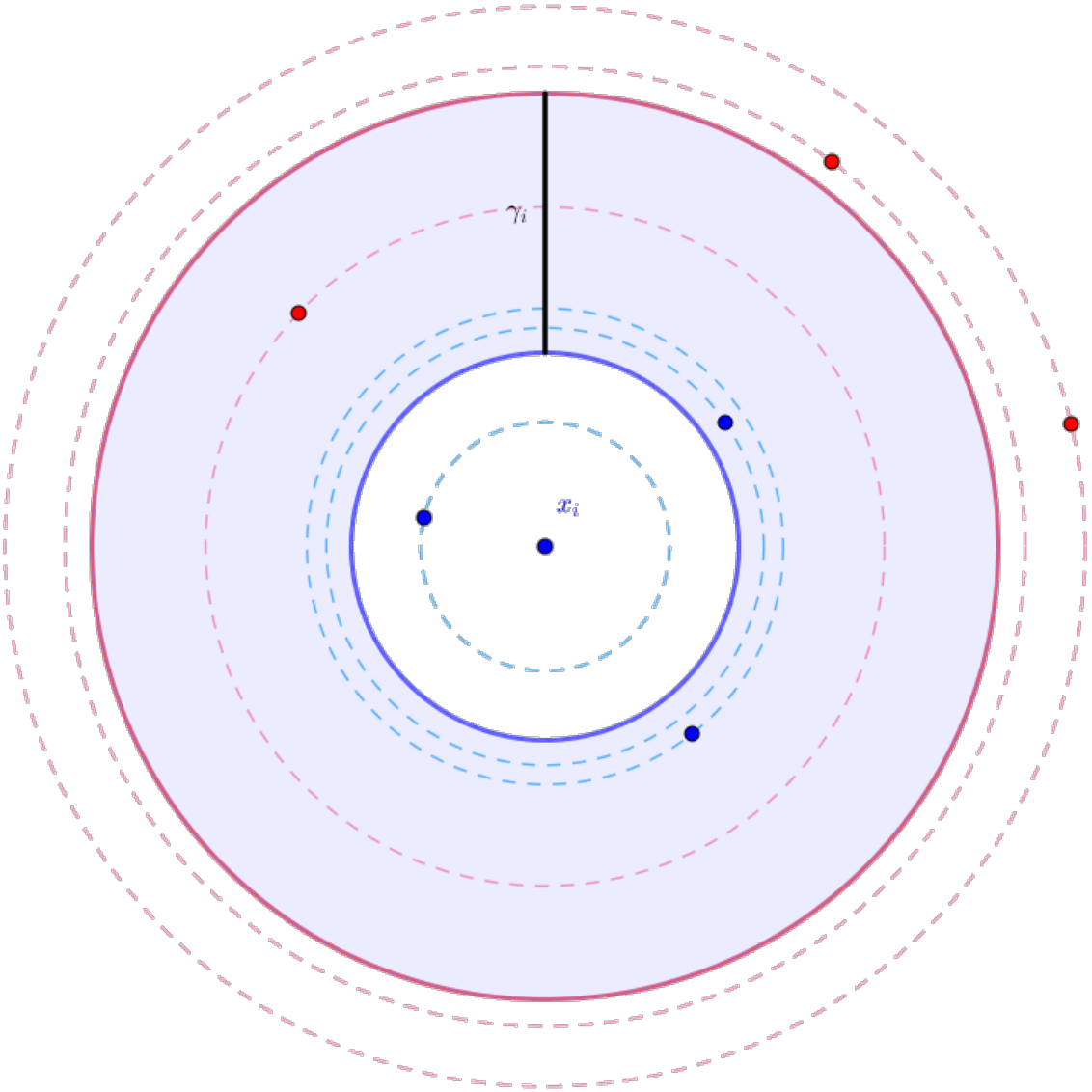


Fig. 1: The geometry of the average neighborhood margin.

1.4.1 References

Masashi Sugiyama “Dimensionality reduction of multimodal labeled data by local fisher discriminant analysis”. In: Journal of Machine Learning Research, 2007, vol 8, May, pages 1027-1061.

1.5 Large Margin Nearest Neighbors (LMNN)

A distance metric learning algorithm for nearest neighbors classification. It learns a metric that pulls the neighbor candidates (*target_neighbors*) near, while pushes near data from different classes (*impostors*) out of the target neighbors margin.

Watch the full LMNN documentation here.

1.5.1 Images

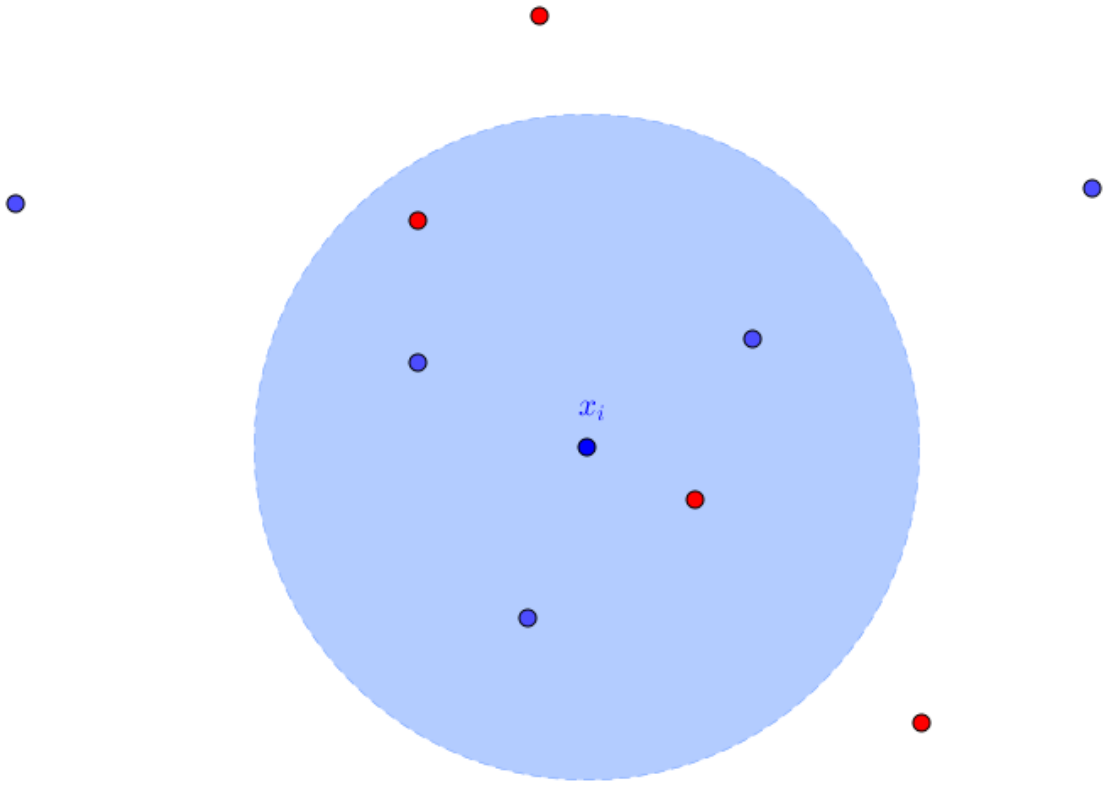


Fig. 2: Impostors and target neighbors

1.5.2 References

Kilian Q Weinberger and Lawrence K Saul. “Distance metric learning for large margin nearest neighbor classification”. In: Journal of Machine Learning Research 10.Feb (2009), pages 207-244.

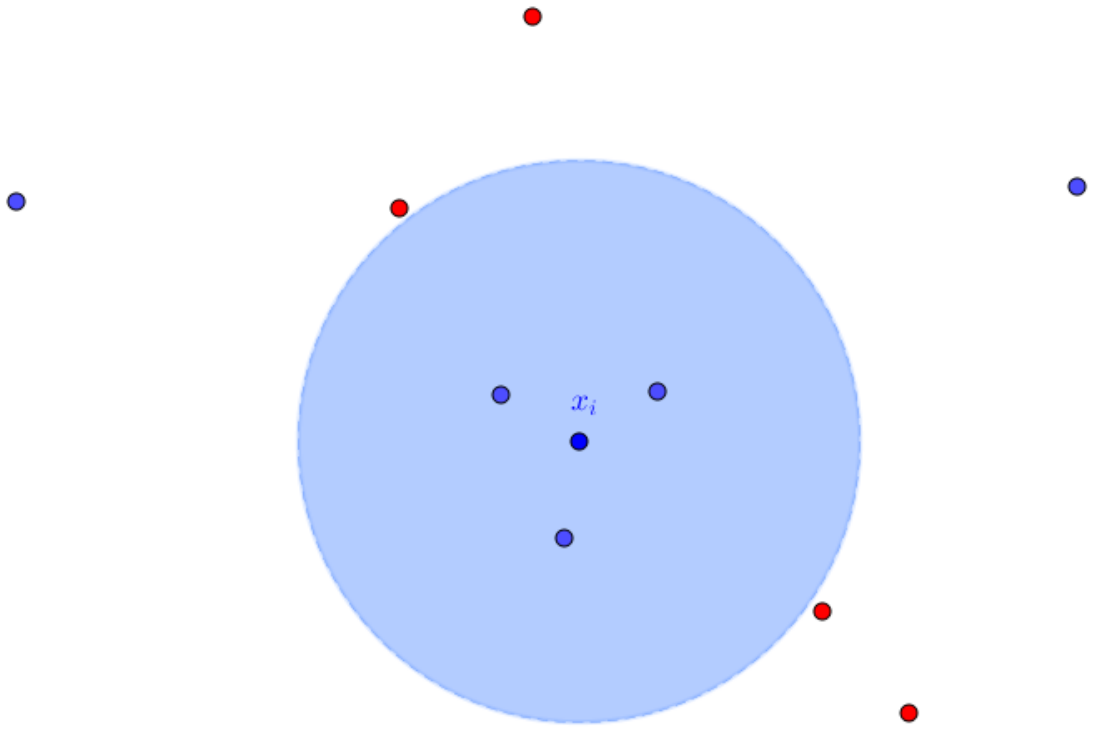


Fig. 3: Data geometry after pulling target neighbors and pushing impostors.

1.6 Neighborhood Component Analysis (NCA)

A distance metric learning for nearest neighbors classification. It learns a transformer that optimizes the expected Leave One Out validation score over the training set.

Watch the full NCA documentation here.

1.6.1 References

Jacob Goldberger et al. "Neighbourhood components analysis". In: Advances in neural information processing systems. 2005, pages 513-520.

1.7 Nearest Class Mean Metric Learning (NCMML)

A distance metric learning algorithm for nearest class mean (NCM) classification. It learns a transformation that optimizes the NCM expected score.

Watch the full NCMML documentation here.

1.7.1 References

Thomas Mensink et al. "Metric learning for large scale image classification: Generalizing to new classes at near-zero cost". In: Computer Vision–ECCV 2012. Springer, 2012, pages 488-501.

1.8 Nearest Class with Multiple Centroids (NCMC)

A distance metric learning algorithm for nearest centroids classification. It learns a transformation that optimizes the expected score of multiple centroid classification. The associated classifier establishes a variable number of centroids for each class via k-Means, and predicts the new labels according to the class of the nearest centroid. This classifier is also available in this package.

Watch the full NCMC documentation here. Watch also the NCMC Classifier documentation.

1.8.1 References

Thomas Mensink et al. "Metric learning for large scale image classification: Generalizing to new classes at near-zero cost". In: Computer Vision–ECCV 2012. Springer, 2012, pages 488-501.

1.9 Information Theoretic Metric Learning (ITML)

An information-theory based distance metric learning algorithm. Given an initial metric, it learns the nearest metric that satisfies some similarity and dissimilarity constraints. The closeness between the metrics is measured using the Kullback-Leibler divergence between the corresponding gaussians.

Watch the full ITML documentation here.

1.9.1 References

Jason V Davis et al. “Information-theoretic metric learning”. In: Proceedings of the 24th international conference on Machine learning. ACM. 2007, pages 209-216.

1.10 Distance Metric Learning through the Maximization of the Jeffrey Divergence (DMLMJ)

An information-theory based distance metric learning algorithm. It learns a transformation that maximizes the Jeffrey divergence between the gaussian distributions associated to the difference spaces for same-class neighbors and different-class neighbors, respectively. This algorithm is also useful for dimensionality reduction.

Watch the full DMLMJ documentation here.

1.10.1 References

Bac Nguyen, Carlos Morell and Bernard De Baets. “Supervised distance metric learning through maximization of the Jeffrey divergence”. In: Pattern Recognition 64 (2017), pages 215-225.

1.11 Maximally Collapsing Metric Learning (MCML)

An information-theory based distance metric learning algorithm. It obtains a metric that minimizes the Kullback-Leibler divergence to the ideal distribution where every points in the same class collapse into a single point, and different class points are infinitely far away.

Watch the full MCML documentation here.

1.11.1 References

Amir Globerson and Sam T Roweis. “Metric learning by collapsing classes”. In: Advances in neural information processing systems. 2006, pages 451-458.

1.12 Learning with Side Information (LSI)

Also known as MMC (*Mahalanobis Metric for Clustering*), this distance metric learning learns a metric that globally minimizes the distance between similar points, with the constraint that dissimilar points must be far enough. This algorithm can be used for supervised learning, but is also valid for clustering with side information.

Watch the full LSI documentation here.

1.12.1 References

Eric P Xing et al. “Distance metric learning with application to clustering with side-information”. In: Advances in neural information processing systems. 2003, pages 521-528.

1.13 Distance Metric Learning with Eigenvalue Optimization (DML-eig)

A global distance metric learning algorithm. It proposes an optimization problem alternative to LSI that can be solved with eigenvalue optimization.

Watch the full DML-eig documentation here.

1.13.1 References

Yiming Ying and Peng Li. "Distance metric learning with eigenvalue optimization". In: Journal of Machine Learning Research 13.Jan (2012), pages 1-26.

1.14 Logistic Discriminant Metric Learning (LDML)

A distance metric learning algorithm that maximizes the likelihood of a logistic based probability distribution.

Watch the full LDML documentation here.

1.14.1 References

Matthieu Guillaumin, Jakob Verbeek and Cordelia Schmid. "Is that you? Metric learning approaches for face identification". In: Computer Vision, 2009 IEEE 12th international conference on. IEEE. 2009, pages 498-505.

1.15 Kernel Large Margin Nearest Neighbors (KLMNN)

The kernelized version of LMNN.

Watch the full KLMNN documentation here.

1.15.1 References

- Kilian Q Weinberger and Lawrence K Saul. "Distance metric learning for large margin nearest neighbor classification". In: Journal of Machine Learning Research 10.Feb (2009), pages 207-244.
- Lorenzo Torresani and Kuang-chih Lee. "Large margin component analysis". In: Advances in neural information processing systems. 2007, pages 1385-1392.

1.16 Kernel Average Neighborhood Margin Maximization (KANMM)

The kernelized version of ANMM.

Watch the full KANMM documentation here.

1.16.1 References

Fei Wang and Changshui Zhang. “Feature extraction by maximizing the average neighborhood margin”. In: Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on. IEEE. 2007, pages 1-8.

1.17 Kernel Distance Metric Learning through the Maximization of the Jeffrey divergence (KDMLMJ)

The kernelized version of DMLMJ.

Watch the full KDMLMJ documentation here.

1.17.1 References

Bac Nguyen, Carlos Morell and Bernard De Baets. “Supervised distance metric learning through maximization of the Jeffrey divergence”. In: Pattern Recognition 64 (2017), pages 215-225.

1.18 Kernel Discriminant Analysis (KDA)

The kernelized version of LDA.

Watch the full KDA documentation here.

1.18.1 References

Sebastian Mika et al. “Fisher discriminant analysis with kernels”. In: Neural networks for signal processing IX, 1999. Proceedings of the 1999 IEEE signal processing society workshop. Ieee. 1999, pages 41-48.

1.19 Kernel Local Linear Discriminant Analysis (KLLDA)

The kernelized version of LLDA.

Watch the full KLMNN documentation here.

1.19.1 References

Masashi Sugiyama “Dimensionality reduction of multimodal labeled data by local fisher discriminant analysis”. In: Journal of Machine Learning Research, 2007, vol 8, May, pages 1027-1061.

1.20 Distance metric learning extensions for some Scikit-Learn classifiers

One of the most important applications of distance metric learning has its focus on similarity learning. Many classifiers use a distance to predicts the labels for new data. Examples of these classifiers are the nearest neighbors classifier and the nearest class mean classifier.

The package pyDML provides an extension of the Scikit-Learn Nearest Neighbors classifier that allows to construct a k-NN classifier that also learns a distance metric using any of the algorithms provided in this package. Watch the documentation here. There is a multi-distance k-NN too.

It is also provided a generalization of the nearest class mean classifier. The Nearest Class with Multiple Centroids classifier computes, for each class, certain number of centroids, using the k-Means clustering algorithm. Then, for predicting new labels, the class of the nearest centroid is assigned. Here again a distance is needed to compute the nearest centroid. Learning an optimal distance is important to improve this classifier. The Nearest Class with Multiple Centroids classifier extends the functionality already provided in Scikit-Learn Nearest Centroid classifier.

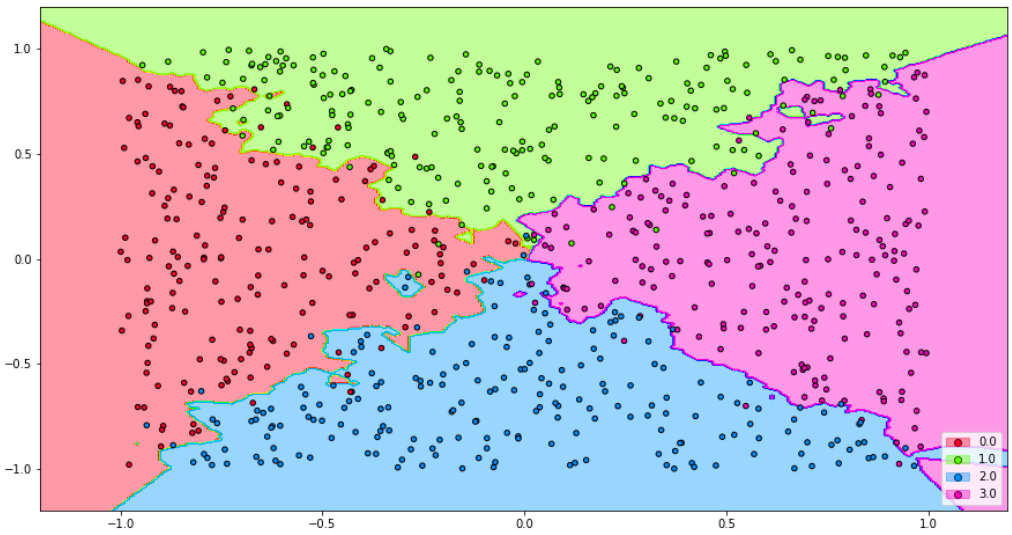
1.21 Distance metric and classifier plots

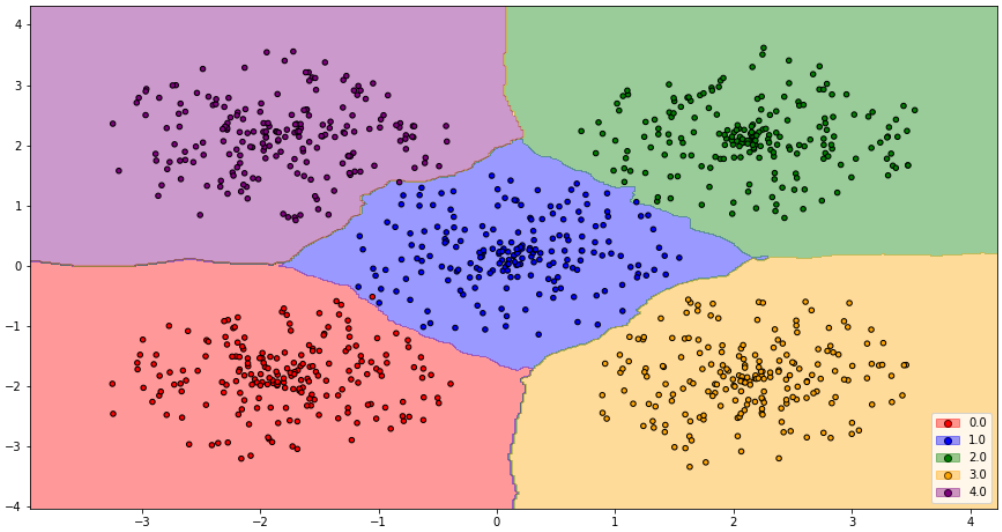
The module dml_plot provides several functions for plotting the regions determined by a classifier and by a distance metric learning combined with a classifier. The classifier_plot function allows to plot the class regions determined by any classifier. With the dml_plot function, a metric can be added to modify the classifier region. This metric can be added via a linear transformation, a metric PSD matrix or a distance metric learning algorithm that will learn it. A special function, when the classifier is the nearest neighbors classifier, is also available. Use in this case knn_plot.

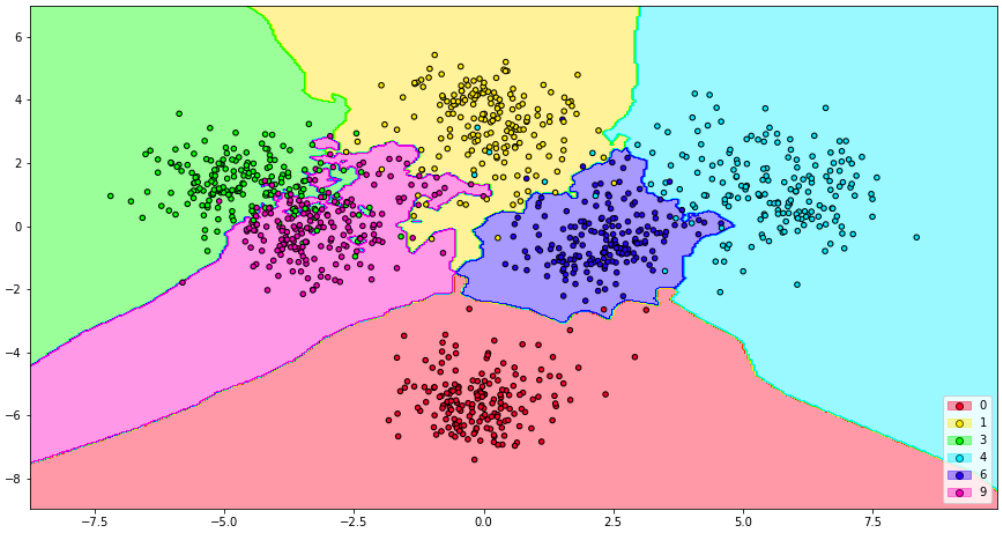
Analogous versions of the previous functions are available to plot simultaneously different pairs of attributes. For this case, watch the functions classifier_pairplots, dml_pairplots and knn_pairplots. Multiple plots, considering different classifiers or distances, can be done with the function dml_multiplot. There is also a 3D classifier plot (still in development), see classifier_plot_3d.

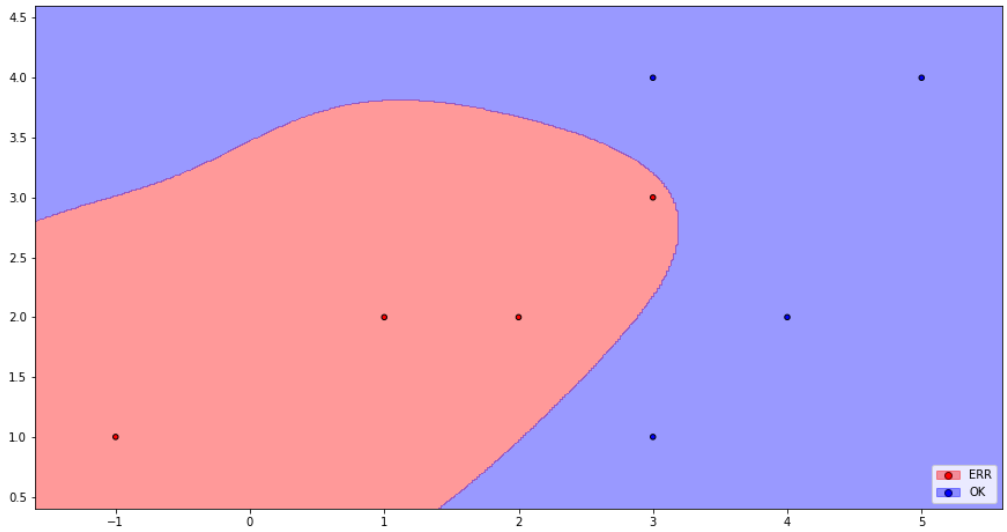
All these libraries use internally Python's matplotlib library.

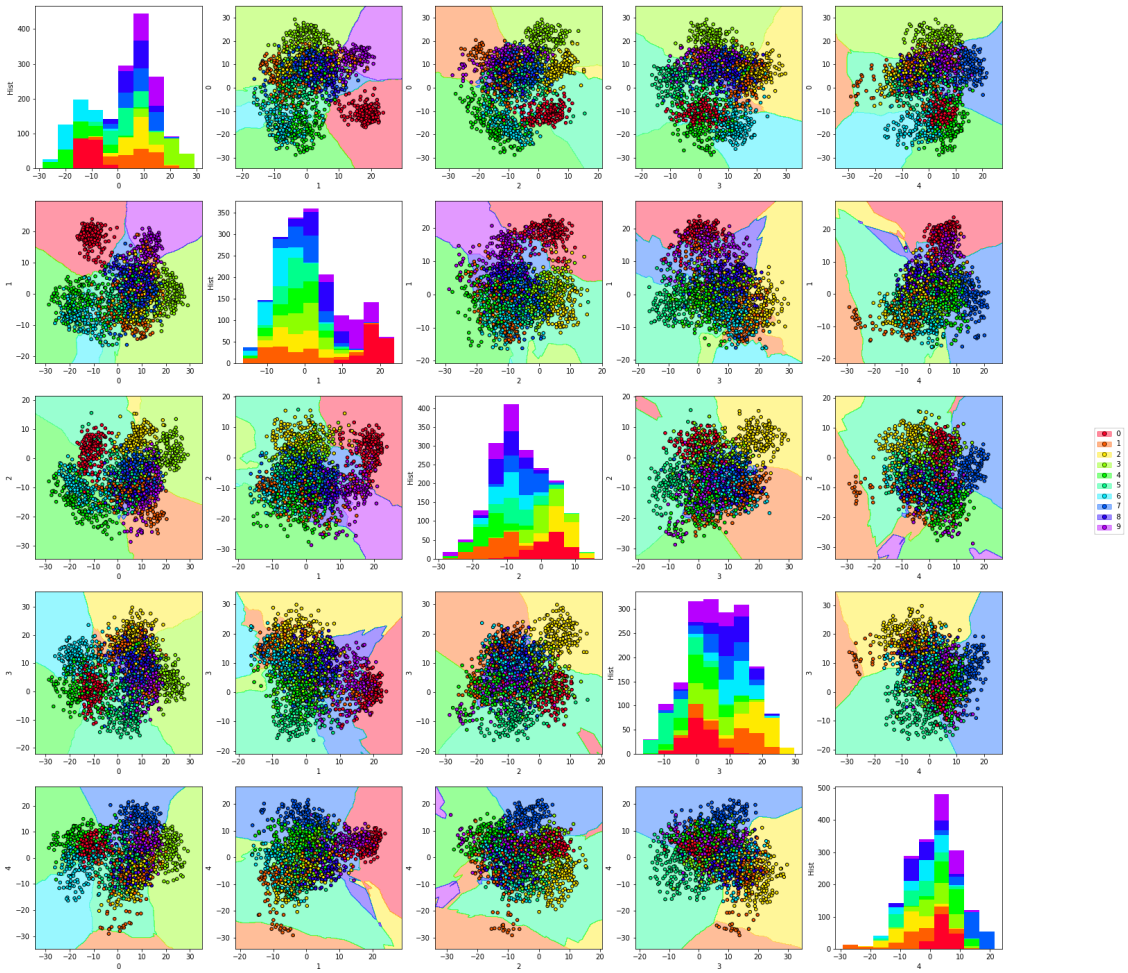
1.21.1 Images

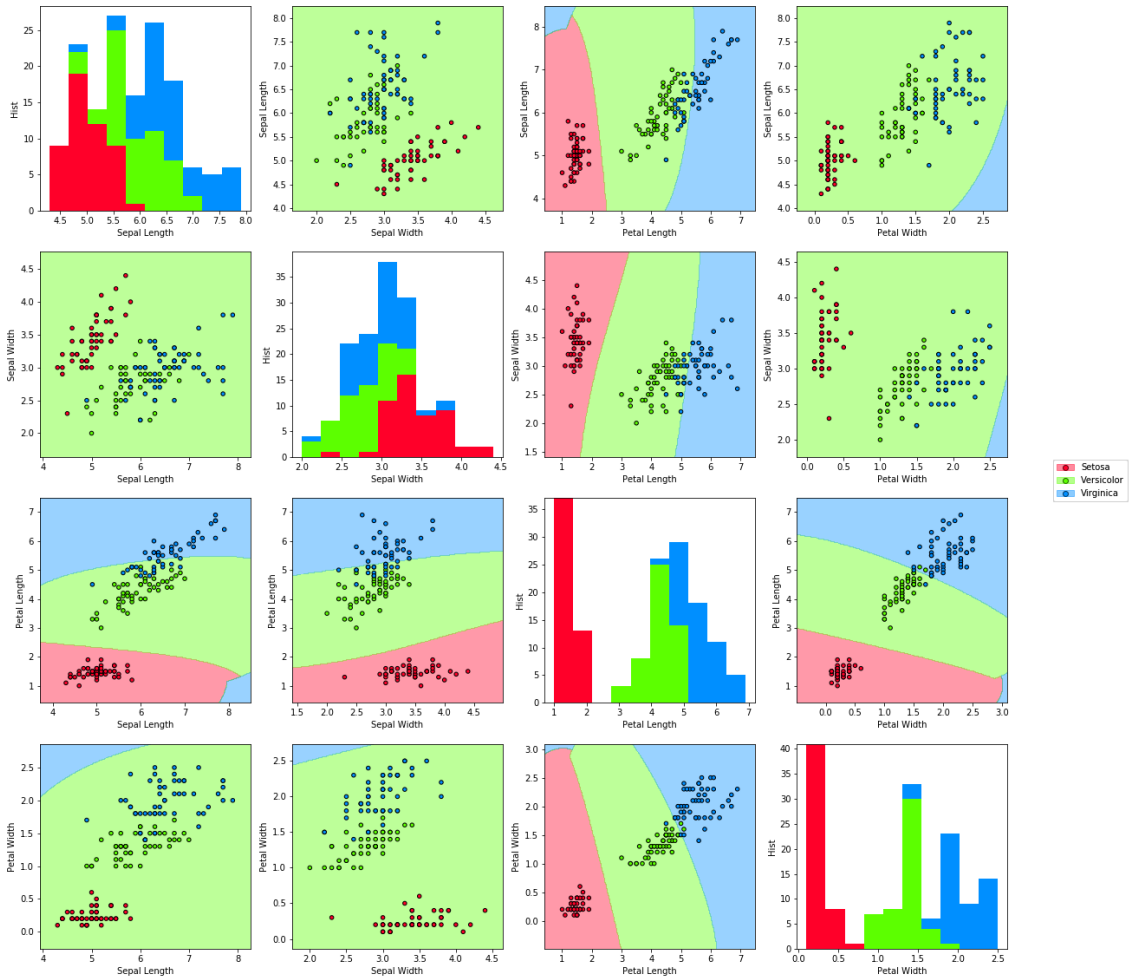



4
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
4
0.0
1.0
2.0
3.0
4.0


6
4
2
0
−2
−4
−6
−8
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
0
1
3
4
6
9


4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
−1
0
1
2
3
4
5
ERR
OK


Hist
0
1
2
3
4
5
6
7
8
9



1.22 Tuning parameters

pyDML package also provides functions for tuning parameters of distance metric algorithms using cross validation. Different scores can be used, from the k-Nearest Neighbors score, for any k, to any of the keys defined in the metadata of the distance metric learning algorithm. See the tune module and the tune and tune_knn functions.

1.23 Package documentation - Indices and tables

- genindex
- modindex

- search

1.24 dml

1.24.1 dml package

Submodules

dml.anmm module

Average Neighborhood Margin Maximization (ANMM)

class dml.anmm.**ANMM**
Bases: *dml.dml_algorithm.DML_Algorithm*

Average Neighborhood Margin Maximization (ANMM)

A DML Algorithm that obtains a transformer that maximizes the distance between the nearest friends and the nearest enemies for each example.

Parameters

num_dims [int, default=None] Dimension desired for the transformed data. If None, all features will be taken.

n_friends [int, default=3] Number of nearest same-class neighbors to compute homogeneus neighborhood.

n_enemies [int, default=1] Number of nearest different-class neighbors to compute heterogeneus neigborhood.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit(*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata(*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transformer(*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

class dml.anmm.**KANMM**
Bases: *dml.dml_algorithm.KernelDML_Algorithm*

The kernelized version of ANMM.

Parameters

num_dims [int, default=None] Dimension desired for the transformed data.

n_friends [int, default=3] Number of nearest same-class neighbors to compute homogeneus neighborhood.

n_enemies [int, default=1] Number of nearest different-class neighbors to compute heterogeneus neigborhood.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit(*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

transformer(*self*)
Obtains the learned projection.

Returns

A [(d'x N) matrix, where d' is the desired output dimension, and N is the number of samples.] To apply A to a new sample x, A must be multiplied by the kernel vector of dimension N obtained by taking the kernels between x and each training sample.

dml.base module

Some basic DML implementations.

Created on Fri Mar 30 19:13:58 2018

@author: jlsuarezdiaz

class dml.base.**Covariance**(*tol=1e-15*, *reg_method='pseudoinverse'*, *alpha=0.001*)
Bases: *dml.dml_algorithm.DML_Algorithm*

A distance metric learning algorithm that learns the distance given by the inverse covariance matrix (the original concept of Mahalanobis distance).

Parameters

tol: float, default=1e-15 The toleration level to consider the covariance matrix invertible.

reg_method: string, default='pseudoinverse' The strategy to deal with singular matrix. Valid options are:

- 'pseudoinverse' : calculates the pseudoinverse matrix when covariance is singular.
- **'addid'** [adds a multiple of identity to the covariance matrix to make it invertible.] The value added is given by the 'alpha' parameter.

alpha [regularization constant to calculate the inverse covariance matrix.] Ignored if reg_method is not 'addid'.

Methods

fit(self, X[, y])	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit(*self*, *X*, *y=None*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples. Added for compatibility, but not used.

Returns

self [object] Returns the instance itself.

metric(*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

class dml.base.**Euclidean**
Bases: *dml.dml_algorithm.DML_Algorithm*

A basic transformer that represents the euclidean distance.

Methods

fit(self, X[, y])	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit(*self*, *X*, *y=None*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metric(*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

transform(*self*, *X=None*)
Applies the metric transformation.

Parameters

X [(N x d) matrix, optional] Data to transform. If not supplied, the training data will be used.

Returns

transformed [(N x d') matrix] Input data transformed to the metric space by $XL^\top$

transformer(*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

class dml.base.**Metric**(*metric*)
Bases: *dml.dml_algorithm.DML_Algorithm*

A DML algorithm that defines a distance given a PSD metric matrix.

Parameters

metric [(d x d) matrix. A positive semidefinite matrix, to define a pseudodistance in euclidean d-dimensional space.]

Methods

fit(self[, X, y])	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit(*self*, *X=None*, *y=None*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metric(*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

class dml.base.**Transformer**(*transformer*)
Bases: *dml.dml_algorithm.DML_Algorithm*

A DML algorithm that defines a distance given a linear transformation.

Parameters

transformer [(d' x d) matrix, representing a linear transformacion from d-dimensional euclidean space] to d'-dimensional euclidean space.

Methods

fit(self[, X, y])	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X=None*, *y=None*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

dml.dml_algorithm module

Distance Metric Algorithm basis.

class dml.dml_algorithm.**DML_Algorithm**
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Abstract class that defines a distance metric learning algorithm. Distance metric learning are implemented as subclasses of DML_Algorithm. A DML Algorithm can compute either a Mahalanobis metric matrix or an associated linear transformation. DML subclasses must override one of the following methods (metric or transformer), according to their computation way.

Methods

fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

metadata (*self*)
Obtains the algorithm metadata. Must be implemented in subclasses.

Returns

dict [A map from string to any.]

metric (*self*)
Computes the Mahalanobis matrix from the transformation matrix. .. math:: M = L^T L

Returns

M [(d x d) matrix. M defines a metric whose distace is given by]

..math:: d(x,y) = sqrt{(x-y)^TM(x-y)}.

transform (*self*, *X=None*)
Applies the metric transformation.

Parameters

X [(N x d) matrix, optional] Data to transform. If not provided, the training data will be used.

Returns

transformed [(N x d') matrix] Input data transformed to the metric space. The learned distance can be measured using the euclidean distance with the transformed data.

transformer (*self*)
Computes a transformation matrix from the Mahalanobis matrix. ..math:: L = M^{1/2}

Returns

L [(d' x d) matrix, with d' <= d. It defines a projection. The distance can be calculated by]

..math:: d(x,y) = |L(x-y)|_2.

class dml.dml_algorithm.**KernelDML_Algorithm**
Bases: dml.dml_algorithm.DML_Algorithm

Abstract class that defines a kernel distance metric learning algorithm. Distance metric learning are implemented as subclasses of KernelDML_Algorithm. A Kernel DML Algorithm can compute a (d' x n) transformer that maps the high dimensional data using the kernel trick. Kernel DML subclasses must override the transformer method, providing the matrix A that performs the kernel trick, that is

$$Lx = A(K(x_1, x), \ldots, K(x_n, x)),$$

where L is the high dimensional transformer and K is the kernel function.

Methods

fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

transform(*self*, *X=None*)
Applies the kernel transformation.

Parameters

X [(N x d) matrix, optional] Data to transform. If not supplied, the training data will be used.

Returns

transformed: (N x d') matrix. Input data transformed by the learned mapping.

dml.dml_eig module

Distance Metric Learning with Eigenvalue Optimization

Created on Fri Mar 9 10:18:35 2018

@author: jlsuarezdiaz

class dml.dml_eig.**DML_eig**
Bases: *dml.dml_algorithm.DML_Algorithm*

Distance Metric Learning with Eigenvalue Optimization (DML-eig)

A DML Algorithm that learns a metric that minimizes the minimum distance between different-class points constrained to the sum of distances at same-class points be non higher than a constant.

Parameters

mu [float, default=1e-4] Smoothing parameter.

tol [float, default=1e-5] Tolerance stop criterion (difference between two point iterations at gradient descent).

eps [float, default=1e-10] Precision stop criterion (norm of gradient at gradient descent).

max_it: int, default=25 Number of iterations at gradient descent.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 9 – continued from previous page

metadata(self)	Obtains algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] initial_error : initial value of the objective error function.

final_error : final value of the objective error function.

metric (*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

dml.dml_plot module

Plot utilies for classifiers and distance metric learners.

Created on Sat Feb 3 16:45:28 2018

@author: jlsuarezdiaz

dml.dml_plot.**classifier_pairplots**(*X, y, clf, attrs=None, xattrs=None, yattrs=None, diag='hist', sections='mean', fitted=False, title=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='center right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=False, **fig_kw*)
This function allows multiple 2D-scatter plots for different pairs of attributes of the same dataset, and to plot regions defined by different classifiers.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

clf [A `ClassifierMixin`.] A classifier. It must support the methods fit(X,y) and predict(X), as specified in `ClassifierMixin`

attrs [List, default=None] A list specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, and xattrs and yattrs are None, all the attributes will be taken.

xattrs [List, default=None] A list specifying the dataset attributes to show in X axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

yattrs [List, default=None] A list specifying the dataset attributes to show in Y axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

`dml.dml_plot.`**`classifier_plot`**(*X, y, clf, attrs=None, sections='mean', fitted=False, f=None, ax=None, title=None, subtitle=None, xrange=None, yrange=None, xlabel=None, ylabel=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='lower right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=True, **fig_kw*)

A 2D-scatter plot for a labeled dataset, together with a classifier that allows to plot each clasiffier region.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

clf [A classifier. It must support the methods fit(X,y) and predict(X), as specified in `ClassifierMixin`]

attrs [List, default=None] A list of two items specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, the two first attributes will be taken.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

subtitle [String, default=None. An optional subtitle for the plot.]

xrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the X axis. If None, it will be calculated according to the maximum and minimum of the X feature.

yrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the Y axis. If None, it will be calculated according to the maximum and minimum of the Y feature.

xlabel [String, default=None. An optional title for the X axis.]

ylabel [String, default=None. An optional title for the Y axis.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_plot.**classifier_plot_3d**(*X, y, clf, attrs=None, sections='mean', fitted=False, f=None, ax=None, elev=0.0, azim=0.0, title=None, subtitle=None, xrange=None, yrange=None, zrange=None, xlabel=None, ylabel=None, zlabel=None, grid_split=[40, 40, 40], grid_step=[0.1, 0.1, 0.1], label_legend=True, legend_loc='lower right', cmap=None, label_colors=None, plot_points=True, plot_regions='all', region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=True, **fig_kw*)

A 3D-scatter plot for a labeled dataset, together with a classifier that allows to plot each clasiffier region.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

clf [A classifier. It must support the methods fit(X,y) and predict(X), as specified in `ClassifierMixin`]

attrs [List, default=None] A list of three items specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, the three first attributes will be taken.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

elev [Float, default=0.0] The elevation parameter for the 3D plot.

azim [Float, default=0.0] The azimut parameter for the 3D plot.

title [String, default=None. An optional title for the plot.]

subtitle [String, default=None. An optional subtitle for the plot.]

xrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the X axis. If None, it will be calculated according to the maximum and minimum of the X feature.

yrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the Y axis. If None, it will be calculated according to the maximum and minimum of the Y feature.

zrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the Z axis. If None, it will be calculated according to the maximum and minimum of the Z feature.

xlabel [String, default=None. An optional title for the X axis.]

ylabel [String, default=None. An optional title for the Y axis.]

zlabel [String, default=None. An optional title for the Z axis.]

grid_split [List, default=[40, 40, 40]] A list with three items, specifying the number of partitions, in the X, Y and Z axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1, 0.1, 0.1]] A list with three items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_plot.**dml_multiplot**(*X, y, nrow=None, ncol=None, ks=None, clfs=None, attrs=None, sections='mean', fitted=False, metrics=None, transformers=None, dmls=None, dml_fitted=False, transforms=None, title=None, subtitles=None, xlabels=None, ylabels=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='center right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=False, **fig_kw*)

This functions allows multiple 2D-scatter plots for a labeled dataset, to plot regions defined by different classifiers and distances. The distances can be provided by a metric PSD matrix, a matrix of a linear transformation, or by a distance metric learning algorithm, that can learn the distance during the plotting, or it can be fitted previously.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

nrow [integer, default=None.] Number of rows of the figure. If any of nrow or ncol is None, it will be generated automatically.

ncol [integer, default=None.] Number of columns of the figure. If any of nrow or ncol is None, it will be generated automatically.

ks [List of int, default=None.] The number of neighbors for the k-NN classifier in each plot. List size must be equal to the number of plots. ks[i] is ignored if clfs[i] is specified.

clfs [List of `ClassifierMixin`. Default=None.] The classifier to use in each plot. List size must be equal to the number of plots.

attrs [List, default=None] A list of two items specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, the two first attributes will be taken.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

metrics [List of Matrix, or 2D-Array. Default=None.] The metric PSD matrix to use in each plot. List size must be equal to the number of plots. metrics[i] is ignored if transformers[i] or dmls[i] are provided.

transformers [List of Matrix, or 2D-Array. Default=None.] A linear transformation to use in each plot. List size must be equal to the number of plots. transformers[i] will be ignored if dmls[i] is provided.

dmls [List of DML_Algorithm, default=None.] A distance metric learning algorithm for each plot. List size must be equal to the number of plots.

dml_fitted [Boolean, default=True.] Specifies if the DML algorithms are already fitted. If True, the algorithms' fit method will not be called.

transforms: List of Boolean, default=True. For each plot where the list item is True, it projects the data by the learned transformer and plots the transform data. Else, the classifier region will be ploted with the original data, but the regions will change according to the learned distance. List size must be equal to the number of plots.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

subtitles [List of String, default=None.] Optional titles for each subplot. List size must be equal to the number of plots.

xlabels [List of String, default=None.] Optional titles for the X axis. List size must be equal to the number of plots.

ylabels [List of String, default=None.] Optional titles for the Y axis. List size must be equal to the number of plots.

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will

define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=False.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_plot.**dml_pairplots**(*X, y, clf, attrs=None, xattrs=None, yattrs=None, diag='hist', sections='mean', fitted=False, metric=None, transformer=None, dml=None, dml_fitted=False, title=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='center right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=False, **fig_kw*)

This function allows multiple 2D-scatter plots for different pairs of attributes of the same dataset, and to plot regions defined by different classifiers and a distance. The distance can be provided by a metric PSD matrix, a matrix of a linear transformation, or by a distance metric learning algorithm, that can learn the distance during the plotting, or it can be fitted previously.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

clf [A `ClassifierMixin`.] A classifier. It must support the methods fit(X,y) and predict(X), as specified in `ClassifierMixin`

attrs [List, default=None] A list specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, and xattrs and yattrs are None, all the attributes will be taken.

xattrs [List, default=None] A list specifying the dataset attributes to show in X axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

yattrs [List, default=None] A list specifying the dataset attributes to show in Y axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

metric [Matrix, or 2D-Array. Default=None.] A positive semidefinite matrix of size (d x d), where d is the number of features. Ignored if dml or transformer is specified.

transformer [Matrix, or 2D-Array. Default=None.] A matrix of size (d' x d), where d is the number of features and d' is the desired dimension. Ignored if dml is specified.

dml [DML_Algorithm, default=None.] A distance metric learning algorithm. If metric, transformer and dml are None, no distances are used in the plot.

dml_fitted [Boolean, default=True.] Specifies if the DML algorithm is already fitted. If True, the algorithm's fit method will not be called.

transform: Boolean, default=True. If True, projects the data by the learned transformer and plots the transform data. Else, the classifier region will be ploted with the original data, but the regions will change according to the learned distance.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_plot.**dml_plot**(*X, y, clf, attrs=None, sections='mean', fitted=False, metric=None, transformer=None, dml=None, dml_fitted=False, transform=True, f=None, ax=None, title=None, subtitle=None, xrange=None, yrange=None, xlabel=None, ylabel=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='lower right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=True, **fig_kw*)

A 2D-scatter plot for a labeled dataset, together with a classifier that allows to plot each clasiffier region, and a distance that can be used by the classifier. The distance can be provided by a metric PSD matrix, a matrix of a linear transformation, or by a distance metric learning algorithm, that can learn the distance during the plotting, or it can be fitted previously.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

clf [A classifier. It must support the methods fit(X,y) and predict(X), as specified in `ClassifierMixin`]

attrs [List, default=None] A list of two items specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, the two first attributes will be taken.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

metric [Matrix, or 2D-Array. Default=None.] A positive semidefinite matrix of size (d x d), where d is the number of features. Ignored if dml or transformer is specified.

transformer [Matrix, or 2D-Array. Default=None.] A matrix of size (d' x d), where d is the number of features and d' is the desired dimension. Ignored if dml is specified.

dml [DML_Algorithm, default=None.] A distance metric learning algorithm. If metric, transformer and dml are None, no distances are used in the plot.

transform: Boolean, default=True. If True, projects the data by the learned transformer and plots the transform data. Else, the classifier region will be ploted with the original data, but the regions will change according to the learned distance.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

subtitle [String, default=None. An optional subtitle for the plot.]

xrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the X axis. If None, it will be calculated according to the maximum and minimum of the X feature.

yrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the Y axis. If None, it will be calculated according to the maximum and minimum of the Y feature.

xlabel [String, default=None. An optional title for the X axis.]

ylabel [String, default=None. An optional title for the Y axis.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_plot.**knn_pairplots**(*X, y, k=1, attrs=None, xattrs=None, yattrs=None, diag='hist', sections='mean', knn_clf=None, fitted=False, metric=None, transformer=None, dml=None, dml_fitted=False, title=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='center right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=False, **fig_kw*)

This function allows multiple 2D-scatter plots for different pairs of attributes of the same dataset, and to plot regions defined by a k-NN classifier and a distance. The distance can be provided by a metric PSD matrix, a matrix of a linear transformation, or by a distance metric learning algorithm, that can learn the distance during the plotting, or it can be fitted previously.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

k [int, default=1.] The number of neighbors for the k-NN classifier. Ignored if knn_clf is specified.

knn_clf [A `ClassifierMixin`.] An already defined kNN classifier. It can be any other classifier, but then options are the same as in :meth: *~dml_plot*.

attrs [List, default=None] A list specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, and xattrs and yattrs are None, all the attributes will be taken.

xattrs [List, default=None] A list specifying the dataset attributes to show in X axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

yattrs [List, default=None] A list specifying the dataset attributes to show in Y axis. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. Ignored if attrs is specified.

diag [String] What to plot on the diagonal subplots. Allowed options are:

- "hist" : An histogram of the data will be plot for the attribute.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

metric [Matrix, or 2D-Array. Default=None.] A positive semidefinite matrix of size (d x d), where d is the number of features. Ignored if dml or transformer is specified.

transformer [Matrix, or 2D-Array. Default=None.] A matrix of size (d' x d), where d is the number of features and d' is the desired dimension. Ignored if dml is specified.

dml [DML_Algorithm, default=None.] A distance metric learning algorithm. If metric, transformer and dml are None, no distances are used in the plot.

dml_fitted [Boolean, default=True.] Specifies if the DML algorithm is already fitted. If True, the algorithm's fit method will not be called.

transform: Boolean, default=True. If True, projects the data by the learned transformer and plots the transform data. Else, the classifier region will be ploted with the original data, but the regions will change according to the learned distance.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot.classifier Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for `suplots()`

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

`dml.dml_plot.`**`knn_plot`**(*X, y, k=1, attrs=None, sections='mean', knn_clf=None, fitted=False, metric=None, transformer=None, dml=None, dml_fitted=False, transform=True, f=None, ax=None, title=None, subtitle=None, xrange=None, yrange=None, xlabel=None, ylabel=None, grid_split=[400, 400], grid_step=[0.1, 0.1], label_legend=True, legend_loc='lower right', cmap=None, label_colors=None, plot_points=True, plot_regions=True, region_intensity=0.4, legend_plot_points=True, legend_plot_regions=True, legend_on_axis=True, **fig_kw*)

A 2D-scatter plot for a labeled dataset to plot regions defined by a k-NN classifier and a distance. The distance can be provided by a metric PSD matrix, a matrix of a linear transformation, or by a distance metric learning algorithm, that can learn the distance during the plotting, or it can be fitted previously.

Parameters

X [array-like of size (N x d), where N is the number of samples, and d is the number of features.]

y [array-like of size N, where N is the number of samples.]

k [int, default=1.] The number of neighbors for the k-NN classifier. Ignored if knn_clf is specified.

knn_clf [A `ClassifierMixin`.] An already defined kNN classifier. It can be any other classifier, but then options are the same as in :meth: *~dml_plot*.

attrs [List, default=None] A list of two items specifying the dataset attributes to show in the scatter plot. The items can be the keys, if X is a pandas dataset, or integer indexes with the attribute position. If None, the two first attributes will be taken.

sections [String, default=fitted] It specifies how to take sections in the features space, if there are more than two features in the dataset. It is used to plot the classifier fixing the non-ploting attributes in this space section. Allowed values are:

'mean' : takes the mean of the remaining attributes to plot the classifier region.

'zeros' : takes the remaining attributes as zeros to plot the classifier region.

fitted [Boolean, default=False.] It indicates if the classifier has already been fitted. If it is false, the function will call the classifier fit method with parameters X,y.

metric [Matrix, or 2D-Array. Default=None.] A positive semidefinite matrix of size (d x d), where d is the number of features. Ignored if dml or transformer is specified.

transformer [Matrix, or 2D-Array. Default=None.] A matrix of size (d' x d), where d is the number of features and d' is the desired dimension. Ignored if dml is specified.

dml [DML_Algorithm, default=None.] A distance metric learning algorithm. If metric, transformer and dml are None, no distances are used in the plot.

dml_fitted [Boolean, default=True.] Specifies if the DML algorithm is already fitted. If True, the algorithm's fit method will not be called.

transform: Boolean, default=True. If True, projects the data by the learned transformer and plots the transform data. Else, the classifier region will be ploted with the original data, but the regions will change according to the learned distance.

f [The :class: *~matplotlib.figure.Figure* object to paint. If None, a new object will be created.]

ax [The :class: *~matplotlib.axes.Axes* object to paint. If None, a new object will be created.]

title [String, default=None. An optional title for the plot.]

subtitle [String, default=None. An optional subtitle for the plot.]

xrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the X axis. If None, it will be calculated according to the maximum and minimum of the X feature.

yrange [List, default=None] A list with two items, specifying the minimum and maximum range to plot in the Y axis. If None, it will be calculated according to the maximum and minimum of the Y feature.

xlabel [String, default=None. An optional title for the X axis.]

ylabel [String, default=None. An optional title for the Y axis.]

grid_split [List, default=[400, 400]] A list with two items, specifying the number of partitions, in the X and Y axis, to make in the plot to paint the classifier region. Each split will define a point where the predict method of the classifier is evaluated. It can be None. In this case, the *grid_step* parameter will be considered.

grid_step [List, default=[0.1,0.1]] A list with two items, specifying the distance between the points in the grid that defines the classifier plot. Each created point in this way will define a point where the predict method of the classifier is evaluated. It is ignored if the parameter *grid_split* is not None.

label_legend [Boolean, default=True. If True, a legend with the labels and its colors will be ploted.]

legend_loc [int, string of a pair of floats, default="lower right".] Specifies the legend position. Ignored if legend is not plotted. Allowed values are: 'best' (0), 'upper right' (1), 'upper left' (2), 'lower left' (3), 'lower right' (4), 'right' (5), 'center left' (6), 'center right' (7), 'lower center' (8), 'upper center' (9), 'center' (10).

Alternatively can be a 2-tuple giving x, y of the lower-left corner of the legend in axes coordinates.

cmap [Colormap, default=None.] A `Colormap` instance or None. If cmap is None and label_colors is None, a default Colormap is used.

label_colors [List, default=None.] A list of size C with matplotlib colors, or strings specitying a color, where C is the number of classes in y. Each class will be plotted with the corresponding color. If cmap is None and label_colors is None, a default Colormap is used.

plot_points [Boolean, default=True.] If True, points will be plotted.

plot_regions [Boolean, default=True.] If True, the classifier regions will be plotted.

region_intensity [Float, default=0.4.] A float between 0 and 1, indicating the transparency of the colors in the classifier regions respect the point colors.

legend_plot_points [Boolean, default=True.] If True, points are plotted in the legend.

legend_plot_regions [Boolean, default=True.] If True, classifier regions are plotted in the legend.

legend_on_axis [Boolean, default=True.] If True, the legend is plotted inside the scatter plot. Else, it is plotted out of the scatter plot.

fig_kw [dict] Additional keyword args for $Matplotlib.suplots$

Returns

f [The plotted :class: *~matplotlib.figure.Figure* object.]

dml.dml_utils module

Utility functions for different DML algoritms

dml.dml_utils.**SDProject**()
Projects a symmetric matrix onto the positive semidefinite cone (considering the Frobenius norm). The projection is made by taking the non negative eigenvalues after diagonalizing.

Parameters

M [2D-Array or Matrix] A symmetric matrix.

Returns

Mplus [2D-Array] The projection of M onto the positive semidefinite cone.

dml.dml_utils.**calc_outers**()
Calculates the outer products between two datasets. All outer products are calculated, so memory may be not enough. To avoid memory errors the output of this function should be used in the input of *calc_outers_i()*.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers [A 4D-array, of shape (N x M x d x d), where outers[i,j] is the outer product between X[i] and Y[j].] It can also be None, if memory was not enough. In this case, outers will be calculated in *calc_outers_i()*.

dml.dml_utils.**calc_outers_i**()
Obtains a subset of outer products from the calculated in *calc_outers()*. If memory was enough, this function just returns a row of outer products from the calculated matrix of outer products. Else, this method calculates this row.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

outers [Numpy array, or None] The output of the function *calc_outers()*.

i [int] The row to fetch from outers, from 0 to N-1.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers_i [A 3D-Array, of shape (M x d x d), where outers_i[j] is the outer product between X[i] and Y[j].] It can also be None, if memory was not enough. In this case, outers will be calculated in *calc_outers_ij()*.

dml.dml_utils.**calc_outers_ij**()
Obtains an outer product between two elements in datasets, from the output calculated in *calc_outers()*.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

outers_i [Numpy array, or None] The output of the function *calc_outers_i()*.

i [int] The row to fetch from outers, from 0 to N-1.

j [int] The column to fetch from outers, from 0 to M-1.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers_i [A 2D-Array, of shape (d x d), with the outer product between X[i] and Y[j].]

dml.dml_utils.**calc_regularized_outers**()
Calculates the outer products between two datasets. All outer products are calculated, so memory may be not enough. To avoid memory errors the output of this function should be used in the input of *calc_outers_i()*.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers [A 4D-array, of shape (N x M x d x d), where outers[i,j] is the outer product between X[i] and Y[j].] It can also be None, if memory was not enough. In this case, outers will be calculated in *calc_outers_i()*.

dml.dml_utils.**calc_regularized_outers_i**()
Obtains a subset of outer products from the calculated in *calc_outers()*. If memory was enough, this function just returns a row of outer products from the calculated matrix of outer products. Else, this method calculates this row.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

outers [Numpy array, or None] The output of the function *calc_outers()*.

i [int] The row to fetch from outers, from 0 to N-1.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers_i [A 3D-Array, of shape (M x d x d), where outers_i[j] is the outer product between X[i] and Y[j].] It can also be None, if memory was not enough. In this case, outers will be calculated in *calc_outers_ij()*.

dml.dml_utils.**calc_regularized_outers_ij**()
Obtains an outer product between two elements in datasets, from the output calculated in *calc_outers()*.

Parameters

X [Numpy array, shape (N x d)] A 2D-array, where N is the number of samples and d is the number of features.

outers_i [Numpy array, or None] The output of the function *calc_outers_i()*.

i [int] The row to fetch from outers, from 0 to N-1.

j [int] The column to fetch from outers, from 0 to M-1.

Y [Numpy array, shape (M x d), default=None] A 2D-array, where M is the number of samples in Y and d is the number of features. If None, Y is taken as X.

Returns

outers_i [A 2D-Array, of shape (d x d), with the outer product between X[i] and Y[j].]

dml.dml_utils.**local_scaling_affinity_matrix**()
Local scaling affinity matrix.

Computes a local scaling affinity matrix A for the dataset (X, y), where .. math:

```
A[i, j] = exp(-\|x_i - x_j\|^2/\sigma_i \sigma_j).
```

The sigma values represent the local scaling of the samples around x_i, and are given by .. math:

```
\sigma_i = \|x_i - x_i^{(K)}\|,
```

where $x_i^{(K)}$ is the K-th nearest neighbor of x_i.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

k [int] The value for the k-th nearest neighbor to consider in the local scaling.

Returns

A [2D-array] The affinity matrix.

dml.dml_utils.**matpack**()
Returns a matrix that takes by columns the elements in the vector v.

Parameters

v [1D-Array] The vector to fit in a matrix.

n [int] The matrix rows.

m [int] The matrix columns.

Returns

A [2D-Array, shape (n x m)] The matrix that takes by columns the elements in v.

dml.dml_utils.**metric_sq_distance**()
Calculates a distance between two points given a metric PSD matrix.

Parameters

M [2D-Array or Matrix] A positive semidefinite matrix defining the distance.

x [Array.] First argument for the distance. It must have the same length as y and the order of M.

y [Array.] Second argument for the distance. It must have the same length as x and the order of M.

dml.dml_utils.**metric_to_linear**()

Converts a metric PSD matrix into an associated linear transformation matrix, so the distance defined by the metric matrix is the same as the euclidean distance after projecting by the linear transformation. This implementation takes the linear transformation corresponding to the square root of the matrix M.

Parameters

M [2D-Array or Matrix] A positive semidefinite matrix.

Returns

L [2D-Array] The matrix associated to the linear transformation that computes the same distance as M.

dml.dml_utils.**neighbors_affinity_matrix**()

Neighbors affinity matrix.

Computes a neighbors affinity matrix A for the dataset (X, y), where A[i, j] = 1 if x_j is a k-nearest neighbor of the same class as x_i, and 0 otherwise.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

k [int] The number of neighbors to consider as nearest neighbors.

Returns

A [2D-array] The affinity matrix.

dml.dml_utils.**pairwise_sq_distances_from_dot**()

Calculates the pairwise squared distance between two datasets given the matrix of dot products.

Parameters

K [2D-Array or Matrix] A matrix with the dot products between two datasets. It verifies ..math:: K[i,j] = langle x_i, y_j rangle

Returns

dists [2D-Array] A matrix with the squared distances between the elements in both datasets. It verifies ..math:: dists[i,j] = d(x_i, y_j)

dml.dml_utils.**unroll**()

Returns a column vector from a matrix with all its columns concatenated.

Parameters

A [2D-Array or Matrix.] The matrix to unroll.

Returns

v [1D-Array] The vector with the unrolled matrix.

dml.dmlmj module

Distance Metric Learning through the Maximization of the Jeffrey divergence (DMLMJ)

Created on Fri Feb 23 12:34:43 2018

@author: jlsuarezdiaz

class dml.dmlmj.**DMLMJ**

Bases: *dml.dml_algorithm.DML_Algorithm*

Distance Metric Learning through the Maximization of the Jeffrey divergence (DMLMJ).

A DML Algorithm that obtains a transformer that maximizes the Jeffrey divergence between the distribution of differences of same-class neighbors and the distribution of differences between different-class neighbors.

Parameters

num_dims [int, default=None] Dimension desired for the transformed data.

n_neighbors [int, default=3] Number of neighbors to consider in the computation of the difference spaces.

alpha [float, default=0.001] Regularization parameter for inverse matrix computation.

reg_tol [float, default=1e-10] Tolerance threshold for applying regularization. The tolerance is compared with the matrix determinant.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit(*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata(*self*)

Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transformer(*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

class dml.dmlmj.**KDMLMJ**
Bases: *dml.dml_algorithm.KernelDML_Algorithm*

The kernelized version of DMLMJ.

Parameters

num_dims [int, default=None] Dimension desired for the transformed data.

n_neighbors [int, default=3] Number of neighbors to consider in the computation of the difference spaces.

alpha [float, default=0.001] Regularization parameter for inverse matrix computation.

reg_tol [float, default=1e-10] Tolerance threshold for applying regularization. The tolerance is compared with the matrix determinant.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit(*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata(*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transformer(*self*)
Obtains the learned projection.

Returns

A [(d'x N) matrix, where d' is the desired output dimension, and N is the number of samples.] To apply A to a new sample x, A must be multiplied by the kernel vector of dimension N obtained by taking the kernels between x and each training sample.

dml.itml module

Information Theoretic Metric Learning (ITML)

Created on Thu Feb 1 17:19:12 2018

@author: jlsuarezdiaz

class dml.itml.**ITML**
Bases: *dml.dml_algorithm.DML_Algorithm*

Information Theoretic Metric Learning (ITML).

A DML algorithm that learns a metric associated to the nearest gaussian distribution satisfying similarity constraints. The nearest gaussian distribution is obtained minimizing the Kullback-Leibler divergence.

Parameters

initial_metric [2D-Array or Matrix] A positive definite matrix that defines the initial metric used to compare.

upper_bound [float, default=None] Bound for dissimilarity constraints. If None, it will be estimated from upper_perc.

lower_bound [float, default=None] Bound for similarity constraints. If None, it will be estimated from lower_perc.

num_constraints [int, default=None] Number of constraints to generate. If None, it will be taken as 40 * k * (k-1), where k is the number of classes.

gamma [float, default=1.0] The gamma value for slack variables.

tol [float, default=0.001] Tolerance stop criterion for the algorithm.

max_iter [int, default=100000] Maximum number of iterations for the algorithm.

low_perc [int, default=5] Lower percentile (from 0 to 100) to estimate the lower bound from the dataset. Ignored if lower_bound is provided.

up_perc [int, default=95] Upper percentile (from 0 to 100) to estimate the upper bound from the dataset. Ignored if upper_bound is provided.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit (*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metric (*self*)

Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

dml.kda module

Kernel Discriminant Analysis (KDA)

Created on Sun Feb 18 18:38:16 2018

@author: jlsuarezdiaz

class dml.kda.**KDA**

Bases: *dml.dml_algorithm.KernelDML_Algorithm*

Kernel Discriminant Analysis (KDA)

Discriminant Analysis in high dimensionality using the kernel trick.

Parameters

solver [string, default='eigen'.]

Solver to use, posible values:

- 'eigen': Eigenvalue decomposition.

n_components [int, default=None.] Number of components (lower than number of classes -1) for dimensionality reduction.

tol [float, default=1e-4] Singularity toleration level.

alpha [float, default=1e-3] Regularization term for singular within-class matrix.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains the algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

transformer (*self*)
Obtains the learned projection.

Returns

A [(d'x N) matrix, where d' is the desired output dimension, and N is the number of samples.] To apply A to a new sample x, A must be multiplied by the kernel vector of dimension N obtained by taking the kernels between x and each training sample.

dml.knn module

k-Nearest Neighbors (kNN)

An interface for kNN adapted to distance metric learning algorithms.

class dml.knn.**kNN**(*n_neighbors*, *dml_algorithm*)
Bases: object

k-Nearest Neighbors (kNN) The nearest neighbors classifier adapted to be used with distance metric learning algorithms.

Parameters

n_neighbors [int] Number of neighbors to consider in classification.

dml_algorithm [DML_Algorithm] The distance metric learning algorithm that will provide the distance in kNN.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
loo_pred(self, X)	Obtains the predicted for the given data using them as a training and with Leave One Out.
loo_prob(self, X)	Predicts the probabilities for the given data using them as a training and with Leave One Out.
loo_score(self, X)	Obtains the score for the given data using them as a training and with Leave One Out.
predict(self[, X])	Predicts the labels for the given data.
predict_orig(self[, X])	Predicts the labels for the given data with the Euclidean distance (with no dml transformations).
predict_proba(self[, X])	Predicts the probabilities for the given data.
predict_proba_orig(self[, X])	Predicts the probabilities for the given data with euclidean distance (with no dml transformations).
score(self[, X, y])	Obtains the classification score for the given data.
score_orig(self[, X, y])	Obtains the classification score for the given data with euclidean distance (with no dml transformation).

fit (*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

loo_pred (*self*, *X*)

Obtains the predicted for the given data using them as a training and with Leave One Out.

X : 2D-Array or Matrix, default=None

The dataset to be used.

Returns

y [1D-Array] The vector with the label predictions.

loo_prob (*self*, *X*)

Predicts the probabilities for the given data using them as a training and with Leave One Out.

X : 2D-Array or Matrix, default=None

The dataset to be used.

Returns

T [2D-Array, shape (N x c)] A matrix with the probabilities for each class. N is the number of samples and c is the number of classes. The element i, j shows the probability of sample X[i] to be in class j.

loo_score (*self*, *X*)

Obtains the score for the given data using them as a training and with Leave One Out.

X : 2D-Array or Matrix, default=None

The dataset to be used.

Returns

score [float] The classification score at kNN. It is calculated as ..math:: card(y_pred == y_real) / n_samples

predict (*self*, *X=None*)

Predicts the labels for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

y [1D-Array] The vector with the label predictions.

predict_orig (*self*, *X=None*)

Predicts the labels for the given data with the Euclidean distance (with no dml transformations). Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

y [1D-Array] The vector with the label predictions.

predict_proba (*self*, *X=None*)

Predicts the probabilities for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

T [2D-Array, shape (N x c)] A matrix with the probabilities for each class. N is the number of samples and c is the number of classes. The element i, j shows the probability of sample X[i] to be in class j.

predict_proba_orig(*self*, *X=None*)

Predicts the probabilities for the given data with euclidean distance (with no dml transformations). Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

T [2D-Array, shape (N x c)] A matrix with the probabilities for each class. N is the number of samples and c is the number of classes. The element i, j shows the probability of sample X[i] to be in class j.

score(*self*, *X=None*, *y=None*)

Obtains the classification score for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

y : 1D-Array, default=None

The real labels for the dataset. It can be None only if X is None.

Returns

score [float] The classification score at kNN. It is calculated as ..math:: card(y_pred == y_real) / n_samples

score_orig(*self*, *X=None*, *y=None*)

Obtains the classification score for the given data with euclidean distance (with no dml transformation). Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

y : 1D-Array, default=None

The true labels for the dataset. It can be None only if X is None.

Returns

score [float] The classification score at kNN. It is calculated as ..math:: card(y_pred == y_real) / n_samples

dml.lda module

Linear Discriminant Analysis (LDA)

class dml.lda.**LDA**

Bases: *dml.dml_algorithm.DML_Algorithm*

Linear Discriminant Analysis (LDA).

A distance metric learning algorithm for supervised dimensionality reduction, maximizing the ratio of variances between classes and within classes. This class is a wrapper for `LinearDiscriminantAnalysis`.

Parameters

num_dims [int, default=None] Number of components (< n_classes - 1) for dimensionality reduction. If None, it will be taken as n_classes - 1. Ignored if thres is provided.

thres [float] Fraction of variability to keep, from 0 to 1. Data dimension will be reduced until the lowest dimension that keeps 'thres' explained variance.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transformer(self)	Obtains the learned projection.

transform

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transform (*self*, *X=None*)

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

dml.ldml module

Logistic Discriminant Metric Learning (LDML)

Created on Mon Mar 12 18:26:53 2018

@author: jlsuarezdiaz

class dml.ldml.**LDML**
Bases: *dml.dml_algorithm.DML_Algorithm*

Logistic Discriminant Metric Learning (LDML).

Distance Metric Learning through the likelihood maximization of a logistic based probability distribution.

Parameters

num_dims [int, default=None.] Number of dimensions for dimensionality reduction. Not supported yet.

b [float, default=1e-3] Logistic function positive threshold.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [float, default=0.3] The initial value for learning rate.

initial_metric [2D-Array or Matrix (d x d), or string, default=None.] If array or matrix, it must be a positive semidefinite matrix with the starting metric for gradient descent, where d is the number of features. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

max_iter [int, default=10] Maximum number of iterations of gradient descent.

prec [float, default=1e-3] Precision stop criterion (gradient norm).

tol [float, default=1e-3] Tolerance stop criterion (difference between two iterations)

descent_method [string, default='SDP'] The descent method to use. Allowed values are:

- 'SDP' : semidefinite programming, consisting of gradient descent with projections onto the PSD cone.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- 'num_iters' : Number of iterations that the descent method took.
- 'initial_error' : Initial value of the objective function.
- 'final_error' : Final value of the objective function.

metric (*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

dml.llda module

Local Linear Discriminant Analysis (LLDA)

class dml.llda.**KLLDA**
Bases: *dml.dml_algorithm.KernelDML_Algorithm*

The kernelized version of LLDA.

Parameters

num_dims [int, default=None] Number of components for dimensionality reduction. If None, it will be taken as n_classes - 1. Ignored if thres is provided.

affinity [array-like or string, default="neighbors"] The affinity matrix, that is, an (N x N) matrix with entries in [0,1], where N is the number of samples, where the (i, j) element specifies

the affinity between samples x_i and x_j. It can be also a string. In this case, the affinity matrix will be computed in the algorithm. Valid strings are:

- **"neighbors"** [An affinity matrix A, where A[i, j] is 1 if x_j is one of the k-nearest neighbors of x_i, will be computed. The value of k] is determined by the 'n_neighbors' attribute.
- **"local-scaling"** [An affinity matrix is computed according to the local scaling method, using the kth-nearest neighbors. The value of k] is determined by the 'n_neighbors' attribute. A recommended value for this case is n_neighbors=7. See [1] for more information.

n_neighbors [int, default=1] Number of neighbors to consider in the affinity matrix. Ignored if 'affinity' is not equal to "neighbors" or "local-scaling".

tol [float, default=1e-4] Singularity toleration level.

alpha [float, default=1e-3] Regularization term for singular within-class matrix.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)

Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transformer(*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

class dml.llda.**LLDA**
Bases: *dml.dml_algorithm.DML_Algorithm*

Local Linear Discriminant Analysis (LDA).

A local version for the Linear Discriminant Analysis.

Parameters

n_components [int, default=None] Number of components for dimensionality reduction. If None, it will be taken as n_classes - 1. Ignored if thres is provided.

affinity [array-like or string, default="neighbors"] The affinity matrix, that is, an (N x N) matrix with entries in [0,1], where N is the number of samples, where the (i, j) element specifies the affinity between samples x_i and x_j. It can be also a string. In this case, the affinity matrix will be computed in the algorithm. Valid strings are:

- **"neighbors"** [An affinity matrix A, where A[i, j] is 1 if x_j is one of the k-nearest neighbors of x_i, will be computed. The value of k] is determined by the 'n_neighbors' attribute.
- **"local-scaling"** [An affinity matrix is computed according to the local scaling method, using the kth-nearest neighbors. The value of k] is determined by the 'n_neighbors' attribute. A recommended value for this case is n_neighbors=7. See [1] for more information.

n_neighbors [int, default=1] Number of neighbors to consider in the affinity matrix. Ignored if 'affinity' is not equal to "neighbors" or "local-scaling".

tol [float, default=1e-4] Singularity toleration level.

alpha [float, default=1e-3] Regularization term for singular within-class matrix.

solver [string, default="sugiyama"] The resolution method. Valid values are:

- "classic" : the original LLDA problem will be computed (building the within-class and between-class matrices in the usual way).
- **"sugiyama"** [the algorithm proposed in [1]. It is faster than the classic method and provides the same results. The solver 'classic'] is kept for testing, but this solver is the recommended one.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 18 – continued from previous page

metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

num_dims : dimension of the reduced data.

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

dml.lmnn module

Large Margin Nearest Neighbors (LMNN)

class dml.lmnn.**KLMNN**
Bases: *dml.dml_algorithm.KernelDML_Algorithm*

The kernelized version of LMNN.

Parameters

num_dims [int, default=None] Desired value for dimensionality reduction. Ignored if solver is 'SDP'.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [float, default=0.3] The initial value for learning rate.

initial_metric [2D-Array or Matrix (d' x d), or string, default=None.] If array or matrix, and solver is SDP, it must be a positive semidefinite matrix with the starting metric (d x d) for gradient descent, where d is the number of features. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

If solver is SGD, then the array or matrix will represent a linear map (d' x d), where d' is the dimension provided in num_dims.

max_iter [int, default=100] Maximum number of iterations of gradient descent.

prec [float, default=1e-8] Precision stop criterion (gradient norm).

tol [float, default=1e-8] Tolerance stop criterion (difference between two iterations)

k [int, default=3] Number of target neighbors to take. If this algorithm is used for nearest neighbors classification, a good choice is to take k as the number of neighbors.

mu [float, default=0.5] The weight of the push error in the minimization algorithm. The objective function is composed of a push error, given by the impostors, with weight mu, and a pull error, given by the target neighbors, with weight (1-mu). It must be between 0.0 and 1.0.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

target_selecion [string, default='kernel'] How to find the target neighbors. Allowed values are:

- 'kernel' : using the euclidean distance in the kernel space.
- 'original' : using the euclidean distance in the original space.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.

Continued on next page

Table 19 – continued from previous page

metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- 'num_iters' : Number of iterations that the descent method took.
- 'initial_error' : Initial value of the objective function.
- 'final_error' : Final value of the objective function.

transformer (*self*)
Obtains the learned projection.

Returns

A [(d'x N) matrix, where d' is the desired output dimension, and N is the number of samples.] To apply A to a new sample x, A must be multiplied by the kernel vector of dimension N obtained by taking the kernels between x and each training sample.

class dml.lmnn.**LMNN**
Bases: *dml.dml_algorithm.DML_Algorithm*, sklearn.base.ClassifierMixin

Large Margin Nearest Neighbors (LMNN)

A distance metric learning algorithm that obtains a metric with target neighbors as near as possible and impostors as far as possible

Parameters

num_dims [int, default=None] Desired value for dimensionality reduction. Ignored if solver is 'SDP'. If NULL, all features will be kept.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [int, default=0.3] The initial value for learning rate. If solver is 'SGD', default value may be too large. In this case it is recommended to use a learning_rate of an order of 1e-3 instead.

initial_metric [2D-Array or Matrix (d' x d), or string, default=None.] If array or matrix, and solver is SDP, it must be a positive semidefinite matrix with the starting metric (d x d) for gradient descent, where d is the number of features. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

If solver is SGD, then the array or matrix will represent a linear map (d' x d), where d' is the dimension provided in num_dims.

max_iter [int, default=100] Maximum number of iterations of gradient descent.

prec [float, default=1e-8] Precision stop criterion (gradient norm).

tol [float, default=1e-8] Tolerance stop criterion (difference between two iterations)

k [int, default=3] Number of target neighbors to take. If this algorithm is used for nearest neighbors classification, a good choice is to take k as the number of neighbors.

mu [float, default=0.5] The weight of the push error in the minimization algorithm. The objective function is composed of a push error, given by the impostors, with weight mu, and a pull error, given by the target neighbors, with weight (1-mu). It must be between 0.0 and 1.0.

soft_comp_interval [int, default=1] Intervals of soft computation. The soft computation relaxes the gradient descent conditions, but makes the algorithm more efficient. This value provides the length of a soft computation interval. After soft_comp_interval iterations of gradient descent, a complete gradient step is performed.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

solver [string, default='SDP'] The algorithm used for minimization. Allowed values are:

- **'SDP'** [semidefinite programming, consisting of gradient descent with projections onto the positive semidefinite cone.] It learns a metric.
- 'SGD' : stochastic gradient descent. It learns a linear transformer.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.

Continued on next page

Table 20 – continued from previous page

predict(self[, X])	Predict the class labels for the provided data, according to the LMNN energy method.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- num_iters : Number of iterations that the descent method took.
- initial_error : Initial value of the objective function.
- final_error : Final value of the objective function.

predict (*self*, *X=None*)
Predict the class labels for the provided data, according to the LMNN energy method.

Parameters

X [array-like, shape (N x d)] Test samples. N is the number of samples and d the number of features. If None, training set will be used.

Returns

y [array of shape (N)] Class labels for each data sample.

dml.lsi module

Learning with Side Information (LSI)

class dml.lsi.**LSI**
Bases: *dml.dml_algorithm.DML_Algorithm*

Learning with Side Information (LSI)

A distance metric learning algorithm that minimizes the sum of distances between similar data, with non similar data constrained to be separated.

Parameters

initial_metric [2D-Array or Matrix (d x d), or string, default=None.] If array or matrix, it must be a positive semidefinite matrix with the starting metric for gradient descent, where d is the number of features. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [float, default=0.1] The initial value for learning rate.

max_iter [int, default=100] Number of iterations for gradient descent.

max_proj_iter [int, default=5000] Number of iterations for iterated projections.

itproj_err [float, default=1e-3] Convergence error criterion for iterated projections

err [float, default=1e-3] Convergence error stop criterion for gradient descent.

supervised [Boolean, default=False] If True, the algorithm will accept a labeled dataset (X,y). Else, it will accept the dataset and the similarity sets, (X,S,D).

Methods

fit(self, X, side)	Fit the model from the data in X and the side information in side
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Obtains the learned metric.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Computes a transformation matrix from the Mahalanobis matrix.

fD	
fD1	
fS	
fS1	
grad_projection	
label_to_similarity_set	

fD(*X*, *D*, *M*, *N*, *d*)

fD1(*X*, *D*, *M*, *N*, *d*, *outers*)

fS(*X*, *S*, *M*, *N*, *d*)

fS1(*X*, *S*, *M*, *N*, *d*, *outers*)

fit (*self*, *X*, *side*)
Fit the model from the data in X and the side information in side

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

side [list of array-like, or 1D-array (N)]

The side information, or the label set. Options:

- side = y, the label set (only if supervised = True)
- side = [S,D], where S is the set of indices of similar data and D is the set of indices of dissimilar data.
- side = [S], where S is the set of indices of similar data. The set D will be the complement of S.

Sets S and D are represented as a boolean matrix (S[i,j]==True iff (i,j) in S)

Returns

——-

self [object] Returns the instance itself.

grad_projection (*grad1*, *grad2*, *d*)

label_to_similarity_set (*y*)

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- 'initial_objective' : Initial value of the objective function.
- 'initial_constraint' : Initial calue of the constraint function.
- 'final_objective' : Final value of the objective function.
- 'final_constraint' : Final value of the constraint function.
- 'iterative_projections_conv_exp' : Convergence ratio, from 0 to 1, of the iterative projections.
- 'projection_iterations_avg' : Average iterations needed in iterative projections.
- 'num_its' : Number of iterations of gradient descent.

metric (*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

dml.mcml module

Maximally collapsing metric learning (MCML)

Created on Mon Mar 12 10:47:23 2018

@author: jlsuarezdiaz

class dml.mcml.**MCML**

Bases: *dml.dml_algorithm.DML_Algorithm*

Maximally Collapsing Metric Learning (MCML)

A distance metric learning algorithm that learns minimizing the KL divergence to the maximally collapsing distribution.

Parameters

num_dims [int, default=None.] Number of dimensions for dimensionality reduction. Not supported yet.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [float, default=0.01] The initial value for learning rate.

initial_metric [2D-Array or Matrix (d x d), or string, default=None.] If array or matrix, it must be a positive semidefinite matrix with the starting metric for gradient descent, where d is the number of features. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

max_iter [int, default=20] Maximum number of iterations of gradient descent.

prec [float, default=1e-3] Precision stop criterion (gradient norm).

tol [float, default=1e-3] Tolerance stop criterion (difference between two iterations)

descent_method [string, default='SDP'] The descent method to use. Allowed values are:

- 'SDP' : semidefinite programming, consisting of gradient descent with projections onto the PSD cone.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Obtains the learned metric.

Continued on next page

Table 22 – continued from previous page

`set_params`(self, **params)	Set the parameters of this estimator.
`transform`(self[, X])	Applies the metric transformation.
`transformer`(self)	Computes a transformation matrix from the Mahalanobis matrix.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- 'num_iters' : Number of iterations that the descent method took.
- 'initial_error' : Initial value of the objective function.
- 'final_error' : Final value of the objective function.

metric (*self*)
Obtains the learned metric.

Returns

M [(dxd) positive semidefinite matrix, where d is the number of features.]

dml.multidml_knn module

Multiple-DML k-Nearest Neighbors (kNN)

class dml.multidml_knn.**MultiDML_kNN**(*n_neighbors*, *dmls=None*, *verbose=False*, ***knn_args*)
Bases: `object`

Multi-DML k-NN

An interface that allows learning k-NN with different distance metric learners simultaneously.

Parameters

n_neighbors [int] The number of neighbors for k-NN.

dmls [list, default=None] A list of distance metric learning algorithms to be learned for k-NN. By default, euclidean distance will be added at the first place of the dml list.

verbose [boolean, default=False] If True, console message about the algorithms execution will be printed.

Methods

add(self, dmls)	Adds a new distance metric learning algorithm to the list.
dmls_string(self)	Obtains the strings with the dml names.
elapsed(self)	Obtains the elapsed time of each DML algorithm
fit(self, X, y)	Fit the model from the data in X and the labels in y.
predict_all(self[, X])	Predicts the labels for the given data.
predict_proba_all(self[, X])	Predicts the probabilities for the given data.
score_all(self[, X, y])	Obtains the scores for the given data.

add(*self*, *dmls*)
Adds a new distance metric learning algorithm to the list.

Parameters

dmls [DML_Algorithm, or list of DMÑ_Algorithm] The DML algorithm or algorithms to add.

dmls_string(*self*)
Obtains the strings with the dml names.

Returns

strings [A list with the names of each dml.]

elapsed(*self*)
Obtains the elapsed time of each DML algorithm

Returns

elapsed [A list of float with the time of each DML.]

fit(*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

predict_all(*self*, *X=None*)
Predicts the labels for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

y [list of 1D-Arrays] A list with the vectors with the label predictions for each DML.

predict_proba_all (*self*, *X=None*)
Predicts the probabilities for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

T [list of 2D-Arrays] A list with the matrices with the label probabilities for each class, for each DML.

score_all (*self*, *X=None*, *y=None*)
Obtains the scores for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

s [list of float] A list with the k-NN scores for each DML.

dml.nca module

Neighbourhood Component Analysis (NCA)

class dml.nca.**NCA**
Bases: *dml.dml_algorithm.DML_Algorithm*

Neighborhood Component Analysis (NCA)

A distance metric learning algorithm that tries to minimize kNN expected error.

Parameters

num_dims [int, default=None] Desired value for dimensionality reduction. If None, the dimension of transformed data will be the same as the original.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [int, default=0.3] The initial value for learning rate.

initial_transform [2D-Array or Matrix (d' x d), or string, default=None.] If array or matrix that will represent the starting linear map for gradient descent, where d is the number of features, and d' is the dimension specified in num_dims. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

max_iter [int, default=100] Maximum number of gradient descent iterations.

prec [float, default=1e-8] Precision stop criterion (gradient norm).

tol [float, default=1e-8] Tolerance stop criterion (difference between two iterations)

descent_method [string, default='SGD'] The descent method to use. Allowed values are:

- 'SGD' : stochastic gradient descent.
- 'BGD' : batch gradient descent.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- num_iters : Number of iterations that the descent method took.
- initial_expectance : Initial value of the objective function (the expected LOO score)
- final_expectance : Final value of the objective function (the expected LOO score)

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

dml.ncmc module

Nearest Class with Multiple Centroids (NCMC)

Created on Wed Feb 28 16:18:39 2018

@author: jlsuarezdiaz

class dml.ncmc.**NCMC**

Bases: *dml.dml_algorithm.DML_Algorithm*

Nearest Class with Multiple Centroids distance metric learner (NCMC).

A distance metric learning algorithm to improve the nearest class with multiple centroids classifier.

Parameters

num_dims [int, default=None] Desired value for dimensionality reduction. If None, the dimension of transformed data will be the same as the original.

centroids_num [int, or list of int, default=3] If it is a list, it must have the same size as the number of classes. In this case, i-th item will be the number of centroids to take in the i-th class. If it is an int, every class will have the same number of centroids.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [int, default=0.3] The initial value for learning rate.

initial_transform [2D-Array or Matrix (d' x d), or string, default=None.] If array or matrix that will represent the starting linear map for gradient descent, where d is the number of features, and d' is the dimension specified in num_dims. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

max_iter [int, default=300] Maximum number of gradient descent iterations.

prec [float, default=1e-15] Precision stop criterion (gradient norm).

tol [float, default=1e-15] Tolerance stop criterion (difference between two iterations)

descent_method [string, default='SGD'] The descent method to use. Allowed values are:

- 'SGD' : stochastic gradient descent.
- 'BGD' : batch gradient descent.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- num_iters : Number of iterations that the descent method took.
- initial_expectance : Initial value of the objective function (the expected score)
- final_expectance : Final value of the objective function (the expected score)

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

class dml.ncmc.**NCMC_Classifier**
Bases: sklearn.base.BaseEstimator, sklearn.base.ClassifierMixin

Nearest Class with Multiple Centroids classifier.

A classifier that makes its predictions by choosing the class who has a centroid the nearest to the point. For each class, an arbitrary number of centroids can be set. This centroids are calculated using k-Means over each class sub-dataset.

Parameters

centroids_num [int, or list of int, default=3] If it is a list, it must have the same size as the number of classes. In this case, i-th item will be the number of centroids to take in the i-th class. If it is an int, every class will have the same number of centroids.

kmeans_args [dictionary] Additional keyword args for k-Means.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predicts the labels for the given data.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

fit (*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

predict (*self*, *X*)

Predicts the labels for the given data. Model needs to be already fitted.

X : 2D-Array or Matrix, default=None

The dataset to be used. If None, the training set will be used. In this case, the prediction will be made using Leave One Out (that is, the sample to predict will be taken away from the training set).

Returns

y [1D-Array] The vector with the label predictions.

dml.ncmml module

Nearest Class Mean Metric Learning (NCMML)

Created on Wed Feb 28 12:07:43 2018

@author: jlsuarezdiaz

class dml.ncmml.**NCMML**

Bases: *dml.dml_algorithm.DML_Algorithm*

Nearest Class Mean Metric Learning (NCMML)

A distance metric learning algorithm to improve the nearest class mean classifier.

Parameters

num_dims [int, default=None] Desired value for dimensionality reduction. If None, the dimension of transformed data will be the same as the original.

learning_rate [string, default='adaptive'] Type of learning rate update for gradient descent. Possible values are:

- 'adaptive' : the learning rate will increase if the gradient step is succesful, else it will decrease.
- 'constant' : the learning rate will be constant during all the gradient steps.

eta0 [int, default=0.3] The initial value for learning rate.

initial_transform [2D-Array or Matrix (d' x d), or string, default=None.] If array or matrix that will represent the starting linear map for gradient descent, where d is the number of features, and d' is the dimension specified in num_dims. If None, euclidean distance will be used. If a string, the following values are allowed:

- 'euclidean' : the euclidean distance.
- 'scale' : a diagonal matrix that normalizes each attribute according to its range will be used.

max_iter [int, default=300] Maximum number of gradient descent iterations.

prec [float, default=1e-15] Precision stop criterion (gradient norm).

tol [float, default=1e-15] Tolerance stop criterion (difference between two iterations)

descent_method [string, default='SGD'] The descent method to use. Allowed values are:

- 'SGD' : stochastic gradient descent.
- 'BGD' : batch gradient descent.

eta_thres [float, default=1e-14] A learning rate threshold stop criterion.

learn_inc [float, default=1.01] Increase factor for learning rate. Ignored if learning_rate is not 'adaptive'.

learn_dec [float, default=0.5] Decrease factor for learning rate. Ignored if learning_rate is not 'adaptive'.

Methods

fit(self, X, y)	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.
metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the metric transformation.
transformer(self)	Obtains algorithm metadata.

fit (*self*, *X*, *y*)

Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is

the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- num_iters : Number of iterations that the descent method took.
- initial_expectance : Initial value of the objective function (the expected score)
- final_expectance : Final value of the objective function (the expected score)

transformer (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:]

- num_iters : Number of iterations that the descent method took.
- initial_expectance : Initial value of the objective function (the expected score)
- final_expectance : Final value of the objective function (the expected score)

dml.pca module

Principal Component Analysis (PCA)

class dml.pca.**PCA**
Bases: *dml.dml_algorithm.DML_Algorithm*

Principal Component Analysis (PCA)

A distance metric learning algorithm for unsupervised dimensionality reduction, obtaining orthogonal directions that maximize the variance. This class is a wrapper for `PCA`.

Parameters

num_dims [int, default=None] Number of components for dimensionality reduction. If None, all the principal components will be taken. Ignored if thres is provided.

thres [float] Fraction of variability to keep, from 0 to 1. Data dimension will be reduced until the lowest dimension that keeps 'thres' explained variance.

Methods

fit(self, X[, y])	Fit the model from the data in X and the labels in y.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
metadata(self)	Obtains algorithm metadata.

Continued on next page

Table 28 – continued from previous page

metric(self)	Computes the Mahalanobis matrix from the transformation matrix.
set_params(self, **params)	Set the parameters of this estimator.
transform(self[, X])	Applies the kernel transformation.
transformer(self)	Obtains the learned projection.

fit (*self*, *X*, *y=None*)
Fit the model from the data in X and the labels in y.

Parameters

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

Returns

self [object] Returns the instance itself.

metadata (*self*)
Obtains algorithm metadata.

Returns

meta [A dictionary with the following metadata:] num_dims : dimension of the reduced data.

acum_eig : eigenvalue rate accumulated in the learned output respect to the total dimension.

transform (*self*, *X=None*)
Applies the kernel transformation.

Parameters

X [(N x d) matrix, optional] Data to transform. If not supplied, the training data will be used.

Returns

transformed: (N x d') matrix. Input data transformed by the learned mapping.

transformer (*self*)
Obtains the learned projection.

Returns

L [(d'xd) matrix, where d' is the desired output dimension and d is the number of features.]

dml.tune module

Tune utilities for distance metric learning.

Created on Fri Feb 9 19:29:06 2018

@author: jlsuarezdiaz

dml.tune.**cross_validate** (*alg*, *X*, *y*, *n_folds=5*, *n_reps=1*, *verbose=False*, *seed=None*)
Cross validation for a classifier.

Parameters

alg [object.] A classifier. It must support the methods fit(X,y) and score(X,y), as specified in `ClassifierMixin`.

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

n_folds [int, default=5] Number of folds for cross validation.

n_reps [int, default=1] Number of cross validations to do.

verbose [boolean, default=False] If True, a console log will be printed.

seed [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` == True.

Returns

results [Pandas Dataframe] A matrix whose rows represent each fold of the cross validation, including also the mean and the std. The columns represent the score, the fit time and the predict time of the classifier.

`dml.tune.`**`metadata_cross_validate`**(*dml*, *X*, *y*, *metrics*, *n_folds=5*, *n_reps=1*, *verbose=False*, *seed=None*, ***knn_args*)

Cross validation for distance metric learning algorithms using metadata as metrics.

Parameters

dml [DML_Algorithm] The distance metric learning algorithm to tune.

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

metrics [list of string and int] The metrics to evaluate. If string, it must be a key of the metadata() function of the DML Algorithm, or 'time'. In this last case, the elapsed fitting time will be returned as a metric. If int, the metric will be the k-NN score, where k is the specified int.

n_folds [int, default=5] Number of folds for cross validation.

n_reps [int, default=1] Number of cross validations to do.

verbose [boolean, default=False] If True, a console log will be printed.

seed [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` == True.

knn_args [dictionary.] Additional keyword arguments for k-NN.

Returns

results [Pandas Dataframe] A matrix whose rows represent each fold of the cross validation, including also the mean and the std. The columns represent the scores of each of the metrics specified.

dml.tune.**tune**(*dml*, *X*, *y*, *dml_params*, *tune_args*, *metrics*, *n_folds=5*, *n_reps=1*, *verbose=False*, *seed=None*, ***knn_args*)

Tune function for a distance metric learning algorithm, allowing as metrics the algorithm metadata, times and k-NN scores.

Parameters

dml [A DML_Algorithm subclass] The distance metric algorithm class to tune.

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

dml_params [dictionary] Additional keyword parameters for the distance metric learning algorithm.

tune_args [dictionary] Parameters of the DML algorithm to tune. Each key has to be a keyword argument of the DML. The associated values have to be lists containing all the desired values for the tuning parameters.

metrics [list of string and int] The metrics to evaluate. If string, it must be a key of the metadata() function of the DML Algorithm, or 'time'. In this last case, the elapsed fitting time will be returned as a metric. If int, the metric will be the k-NN score, where k is the specified int.

n_folds [int, default=5] Number of folds for cross validation.

n_reps [int, default=1] Number of cross validations to do.

verbose [boolean, default=False] If True, a console log will be printed.

seed [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` == True.

knn_args [dictionary.] Additional keyword arguments for k-NN.

Returns

tune_results [Pandas Dataframe] A dataframe whose entries are all the cases considered for the tune parameters, and with a single column that shows the cross validation score for each case.

best_performance [Tuple] A pair with the best case obtained, together with its corresponding score.

best_dml [DML_Algorithm] The DML Algorithm object that obtained the best result in the tuning.

detailed_results [Dictionary] A dictionary whose keys are all the possible cases, and each entry is the cross validation table for the corresponding case, containing the scores for every fold.

dml.tune.**tune_knn**(*dml*, *X*, *y*, *n_neighbors*, *dml_params*, *tune_args*, *n_folds=5*, *n_reps=1*, *verbose=False*, *seed=None*, ***knn_args*)

A tune function for a distance metric learning algorithm, using k-NN score as metric.

Parameters

dml [A DML_Algorithm subclass] The distance metric algorithm class to tune.

X [array-like, shape (N x d)] Training vector, where N is the number of samples, and d is the number of features.

y [array-like, shape (N)] Labels vector, where N is the number of samples.

n_neighbors [int] Number of neighbors for k-NN.

dml_params [dictionary] Additional keyword parameters for the distance metric learning algorithm.

tune_args [dictionary] Parameters of the DML algorithm to tune. Each key has to be a keyword argument of the DML. The associated values have to be lists containing all the desired values for the tuning parameters.

n_folds [int, default=5] Number of folds for cross validation.

n_reps [int, default=1] Number of cross validations to do.

verbose [boolean, default=False] If True, a console log will be printed.

seed [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when `shuffle` == True.

knn_args [dictionary.] Additional keyword arguments for k-NN.

Returns

tune_results [Pandas Dataframe] A dataframe whose entries are all the cases considered for the tune parameters, and with a single column that shows the cross validation score for each case.

best_performance [Tuple] A pair with the best case obtained, together with its corresponding score.

best_dml [DML_Algorithm] The DML Algorithm object that obtained the best result in the tuning.

detailed_results [Dictionary] A dictionary whose keys are all the possible cases, and each entry is the cross validation table for the corresponding case, containing the scores for every fold.

Module contents

The Distance Metric Learning module.

1.25 Applications

1.25.1 Improving similarity learning classifiers

Learning a distance that fits the data properly will improve the accuracy of distance-based classifiers.

1.25.2 Dimensionality reduction

Many of the distance metric learning algorithms can learn projections onto low dimensional spaces. Dimensionality reduction improves the classifier eficiency, reduces overfitting and avoids problems such a the curse of dimensionality present in some similarity classifiers.

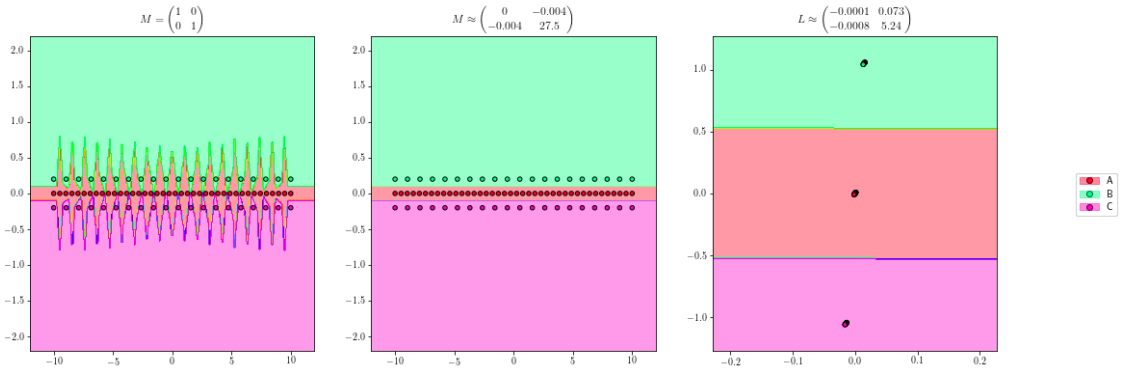


Fig. 4: 1-NN classification with euclidean distance (left), 1-NN classification after learning a distance (center) and the equivalent projection learned (right)

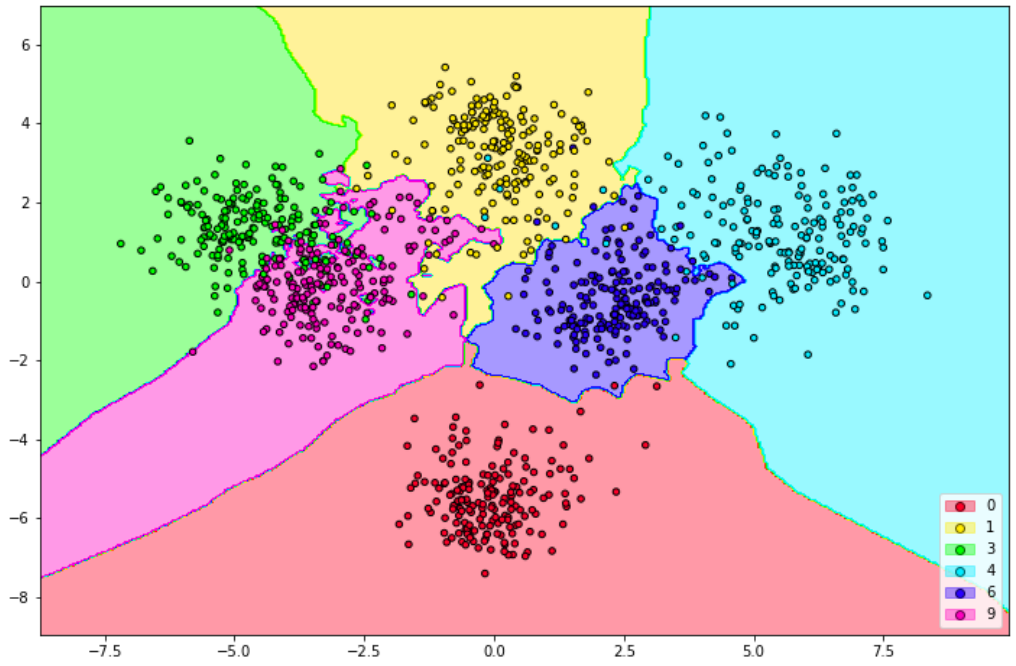


Fig. 5: The digits dataset (64 features) projected onto a plane with a distance metric learning algorithm.

1.26 Examples

1.26.1 Fitting distance metric learning algorithms

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris

>>> # Loading DML Algorithm
>>> from dml import NCA

>>> # Loading dataset
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']

>>> # DML construction
>>> nca = NCA()

>>> # Fitting algorithm
>>> nca.fit(X,y)

>>> # We can look at the algorithm metadata after fitting it
>>> meta = nca.metadata()
>>> meta
{'final_expectance': 0.95771240234375,
 'initial_expectance': 0.8380491129557291,
 'num_iters': 3}

>>> # We can see the metric the algorithm has learned.
>>> # This metric is the PSD matrix that defines how the distance is measured:
>>> # d(x,y) = (x-y).T.dot(M).dot(x-y)
>>> M = nca.metric()
>>> M
array([[ 1.19098678,  0.51293714, -2.15818151, -2.01464351],
       [ 0.51293714,  1.58128238, -2.14573777, -2.10714773],
       [-2.15818151, -2.14573777,  6.46881853,  5.86280474],
       [-2.01464351, -2.10714773,  5.86280474,  6.83271473]])

>>> # Equivalently, we can see the learned linear map.
>>> # The distance coincides with the euclidean distance after applying the linear␣
↪map.
>>> L = nca.transformer()
>>> L
array([[ 0.77961001, -0.01911998, -0.35862791, -0.23992861],
       [-0.04442949,  1.00747788, -0.29936559, -0.25812144],
       [-0.60744415, -0.57288453,  2.16095076,  1.35212555],
       [-0.46068713, -0.48755353,  1.25732916,  2.20913531]])

>>> # Finally, we can obtain the transformed data ...
>>> Lx = nca.transform()
>>> Lx[:5,:]
array([[ 3.35902632,  2.8288461 , -1.80730485, -1.85385382],
       [ 3.21266431,  2.33399305, -1.39937375, -1.51793964],
       [ 3.0887811 ,  2.57431109, -1.60855691, -1.64904583],
       [ 2.94100652,  2.41813313, -1.05833389, -1.30275593],
       [ 3.27915332,  2.93403684, -1.80384889, -1.85654046]])
```

(continues on next page)

(continued from previous page)

```
>>> # ... or transform new data.
>>> X_ = np.array([[1.0,0.0,0.0,0.0],[1.0,1.0,0.0,0.0],[1.0,1.0,1.0,0.0]])
>>> Lx_ = nca.transform(X_)
>>> Lx_
array([[ 0.77961001, -0.04442949, -0.60744415, -0.46068713],
       [ 0.76049003,  0.9630484 , -1.18032868, -0.94824066],
       [ 0.40186212,  0.66368281,  0.98062208,  0.3090885 ]])
```

1.26.2 Similarity learning classifier extensions for Scikit-learn

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris

>>> from dml import NCA, kNN, MultiDML_kNN

>>> # Loading dataset
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']

>>> # Initializing transformer and predictor
>>> nca = NCA()
>>> knn = kNN(n_neighbors=7,dml_algorithm=nca)

>>> # Fitting transformer and predictor
>>> nca.fit(X,y)
>>> knn.fit(X,y)

# Now we can predict the labels for k-NN with the learned distance.
>>> knn.predict() # Also we can use predict(X_) for other datasets.
>>>               # When using the training set predictions are made
>>>               # leaving the sample to predict out.
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  2.,  1.,  2.,  1.,  1.,  1.,  1.,  2.,
        1.,  1.,  1.,  1.,  1.,  2.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,
        2.,  2.,  2.,  2.,  2.,  2.,  2.])

>>> knn.predict_proba()[-10:,:] # Again it can be used for other datasets.
array([[ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.14285714,  0.85714286],
       [ 0.        ,  0.        ,  1.        ],
```

(continues on next page)

(continued from previous page)

```
       [ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  0.14285714,  0.85714286]])

>>> knn.score() # The classification score (score(X_,y_) for other datasets).
0.97333333333333338

>>> # We can also compare with the euclidean distance k-NN
>>> knn.score_orig()
0.96666666666666667

>>> # With MultiDML_kNN we can test multiple dmls. In this case, dmls are fitted␣
↪automatically.
>>> lda = LDA()
>>> mknn = MultiDML_kNN(n_neighbors=7,dmls=[lda,nca])
>>> mknn.fit(X,y)

>>> # And we can predict and take scores in the same way, for every dml.
>>> # The euclidean distance will be added always in first place.
>>> mknn.score_all() # It will show [euclidean, lda, nca]
array([ 0.96666667,  0.96666667,  0.97333333])

>>> # The NCMC Classifier works like every ClassifierMixin.
>>> ncmc = NCMC_Classifier(centroids_num=2)
>>> ncmc.fit(X,y)
>>> ncmc.score(X,y)
0.95333333333333337

>>> # To learn a distance to use with NCMC Classifier, and with any other distance␣
↪classifier
>>> # we can use pipelines.
>>> from sklearn.pipeline import Pipeline
>>> dml_ncmc = Pipeline([('nca',nca),('ncmc',ncmc)])
>>> dml_ncmc.fit(X,y)
>>> dml_ncmc.score(X,y)
0.97999999999999998
```

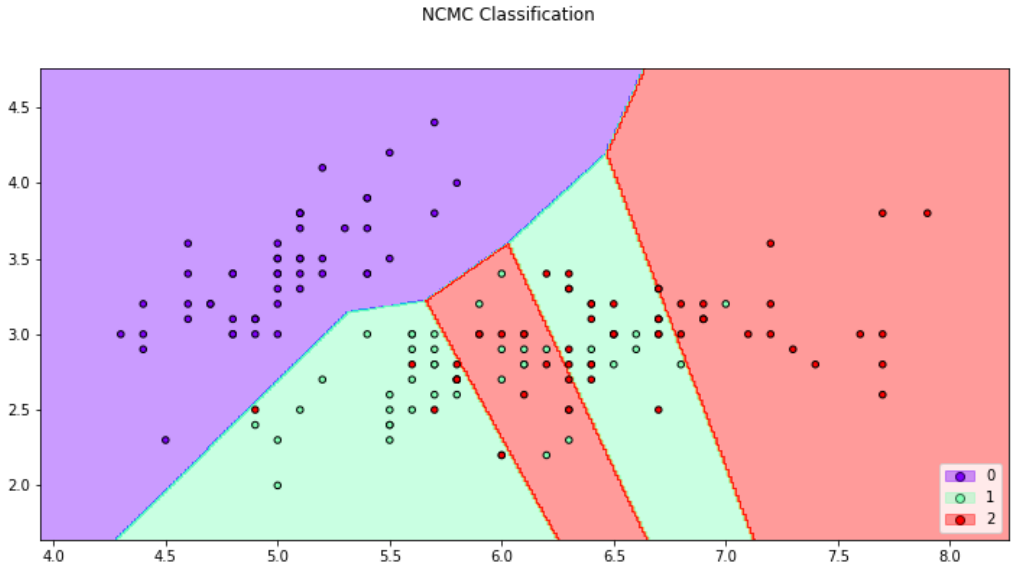
1.26.3 Plotting classifier regions induced by different distances

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from dml import NCA, LDA, NCMC_Classifier, classifier_plot, dml_plot, knn_plot,
>>>                 dml_multiplot, knn_pairplots

>>> # Loading dataset
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']

>>> # Initializing transformers and predictors
>>> nca = NCA()
>>> lda = LDA()
>>> ncmc = NCMC_Classifier(centroids_num=2)

>>> # We can plot regions for different classifiers
>>> f1 = classifier_plot(X[:,[0,1]],y,clf=ncmc,title = "NCMC Classification",
```

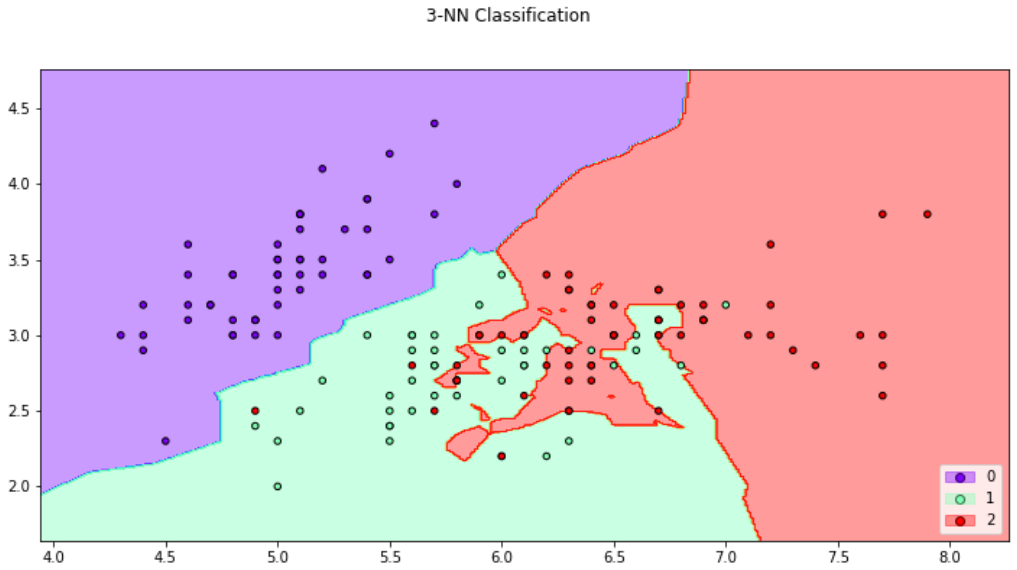
(continues on next page)

(continued from previous page)

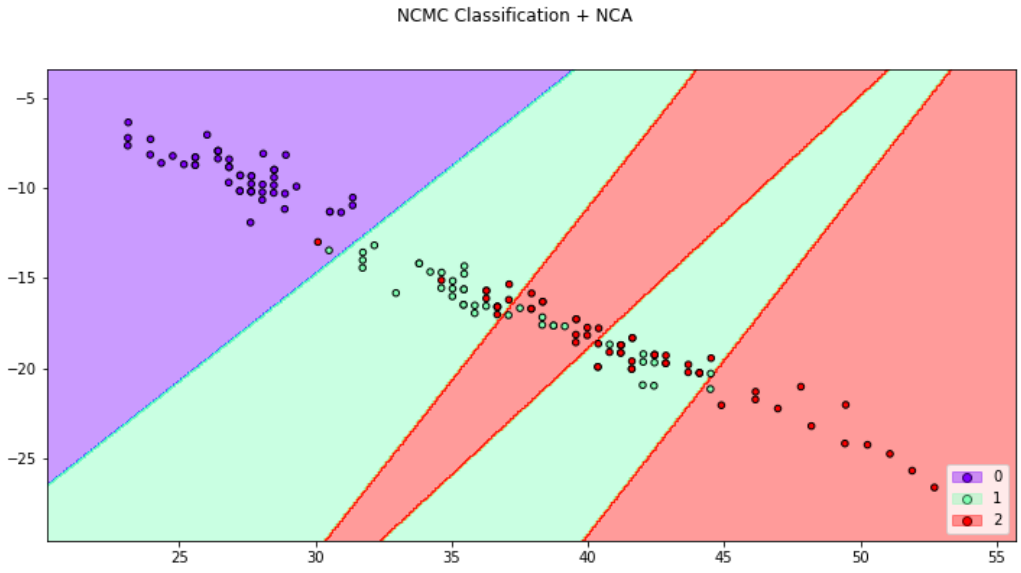
```
>>>                          cmap="rainbow",figsize=(12,6))
```



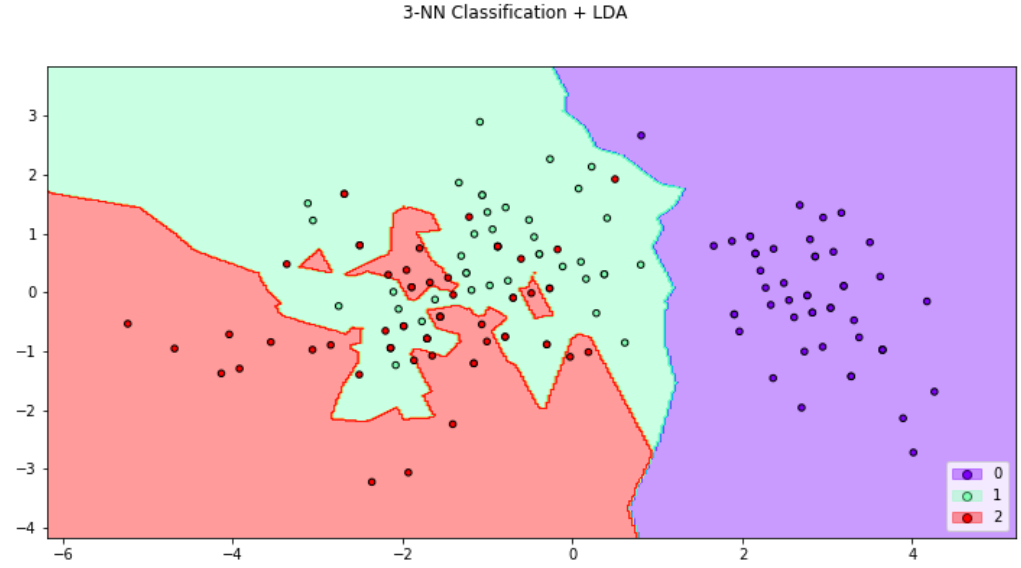

```
>>> f2 = knn_plot(X[:,[0,1]],y,k=3,title = "3-NN Classification", cmap="rainbow",
>>>                 figsize=(12,6))
```



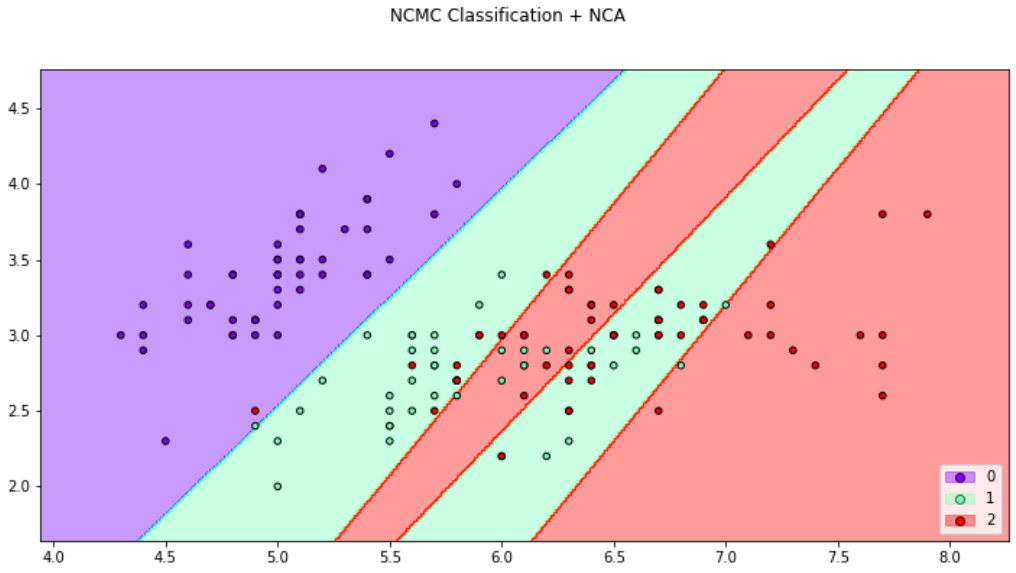

```
>>> # We can also make with the transformation determined by a metric,
>>> # a transformer or a DML Algorithm
>>> f3 = dml_plot(X[:,[0,1]],y,clf=ncmc,dml=nca,title = "NCMC Classification + NCA",
>>>                 cmap="rainbow",figsize=(12,6))
```



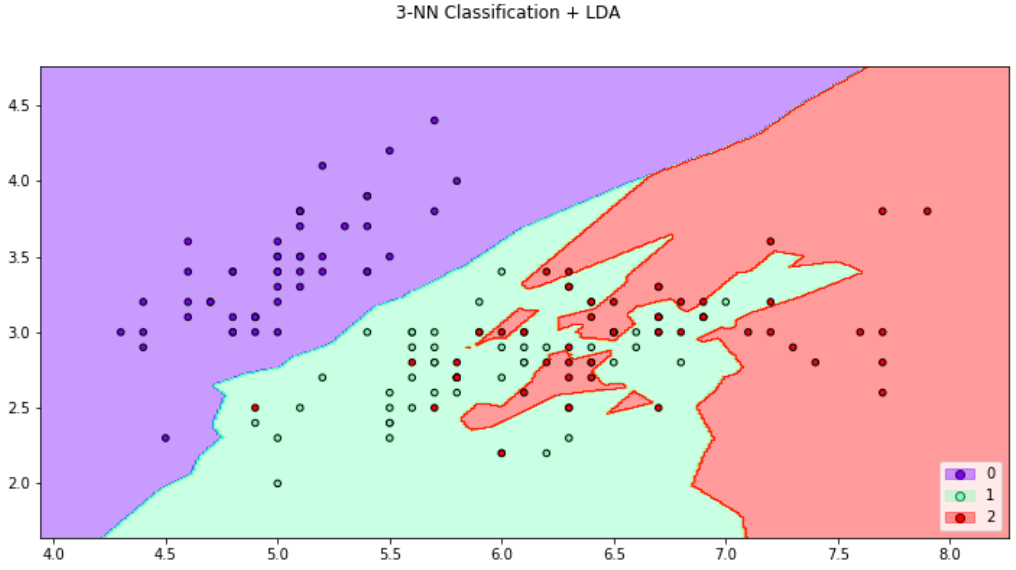

```
>>> f4 = knn_plot(X[:,[0,1]],y,k=2,dml=lda,title="3-NN Classification + LDA",
>>>                cmap="rainbow",figsize=(12,6))
```



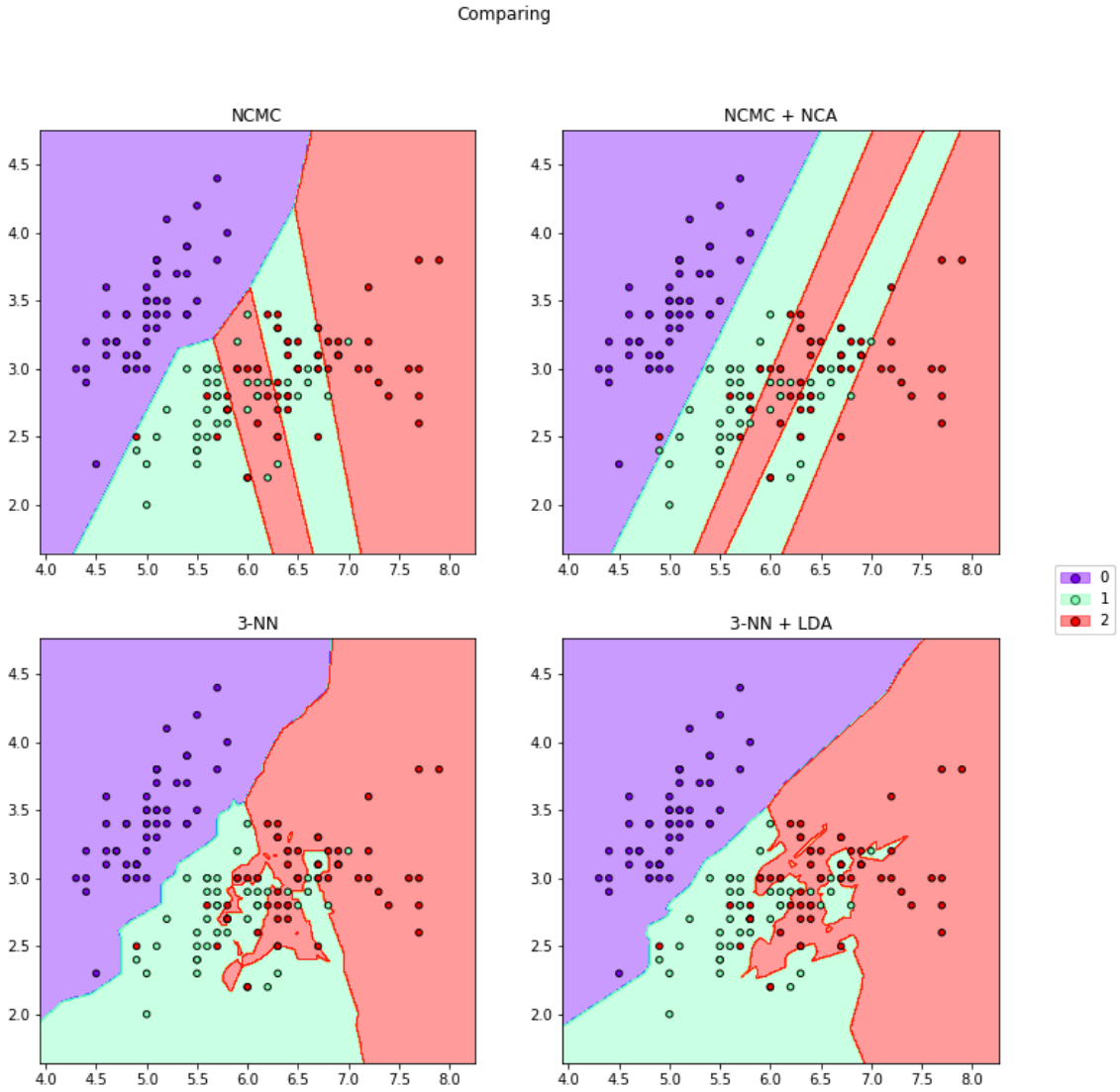

```
>>> # Or we can see how the distance changes the classifier region
>>> # using the option transform=False
>>> f5 = dml_plot(X[:,[0,1]],y,clf=ncmc,dml=nca,title = "NCMC Classification + NCA",
>>>                cmap="rainbow",transform=False,figsize=(12,6))
```



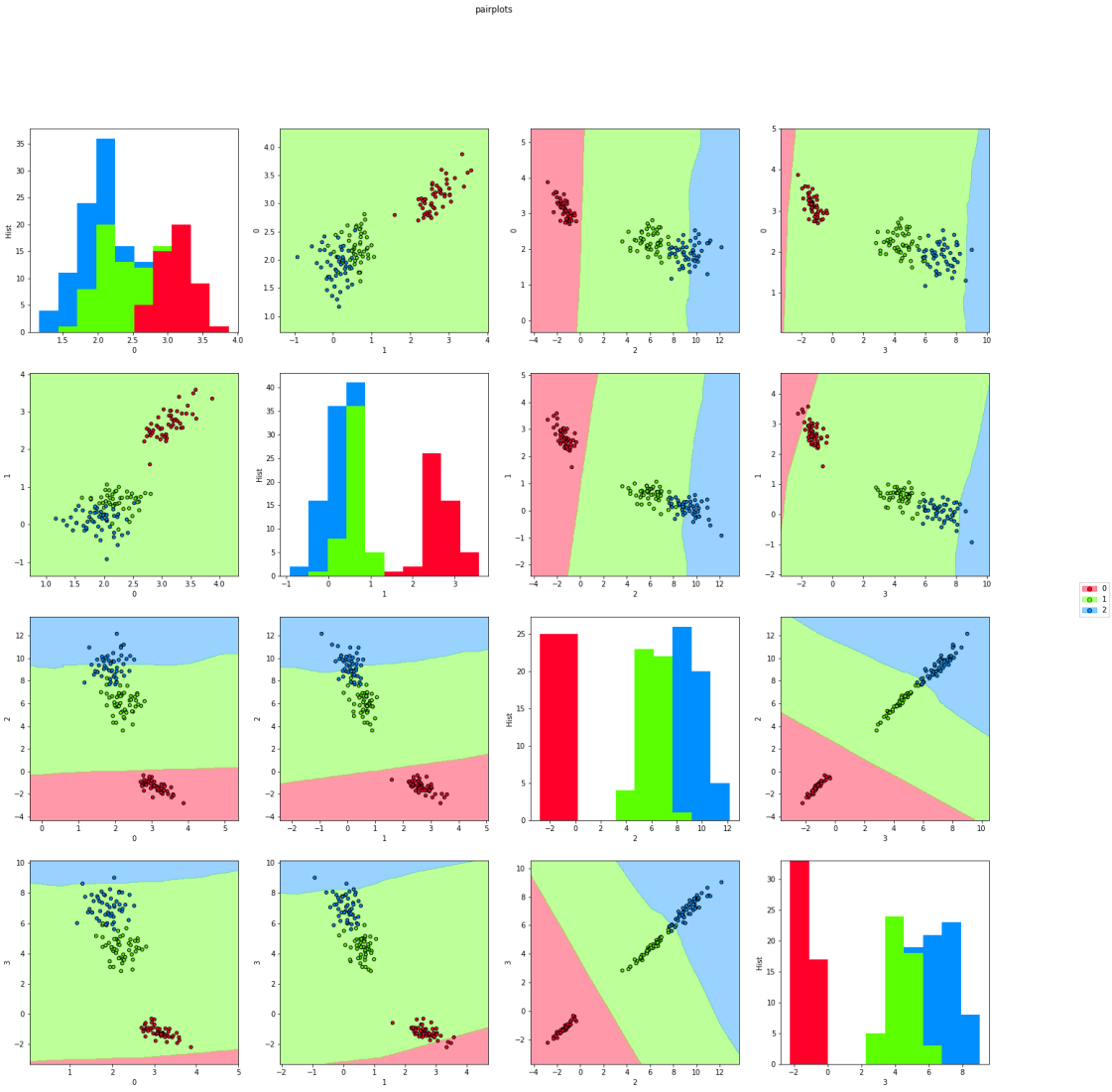

```
>>> f6 = knn_plot(X[:,[0,1]],y,k=2,dml=lda,title="3-NN Classification + LDA",
>>>               cmap="rainbow",transform=False,figsize=(12,6))
```




```
>>> # We can compare different algorithms or distances together in the same figure
>>> f7 = dml_multiplot(X[:,[0,1]],y,nrow=2,ncol=2,ks=[None,None,3,3],
>>>                    clfs=[ncmc,ncmc,None,None],dmls=[None,nca,None,lda],
>>>                    transforms=[False,False,False,False],title="Comparing",
>>>                    subtitles=["NCMC","NCMC + NCA","3-NN","3-NN + LDA"],
>>>                    cmap="rainbow",figsize=(12,12))
```




```
>>> # Finally, we can also plot each pair of attributes. Here the classifier region
>>> # is made taking a section in the features space.
>>> f8 = knn_pairplots(X,y,k=3,sections="mean",dml=nca,title="pairplots",
>>>                    cmap="gist_rainbow",figsize=(24,24))
```



1.26.4 Tuning parameters

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from dml import NCA, tune

>>> # Loading dataset
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']

>>> # Using cross validation we can tune parameters for the DML algorithms.
>>> # Here, we tune the NCA algorithm, with a fixed parameter learning_rate='constant
↪'.
```

(continues on next page)

(continued from previous page)

```
>>> # The parameters we tune are num_dims and eta0.
>>> # The metrics we use are 3-NN and 5-NN scores, and the final expectance metadata␣
↪of NCA.
>>> # A 5-fold cross validation is done twice, to obtain the results.
>>> results,best,nca_best,detailed = tune(NCA,X,y,dml_params={'learning_rate':
↪'constant'},
>>>                                       tune_args={'num_dims':[3,4],'eta0':[0.001,0.
↪01,0.1]},
>>>                                       metrics=[3,5,'final_expectance'],
>>>                                       n_folds=5,n_reps=2,seed=28,verbose=True)
*** Tuning Case  {'num_dims': 3, 'eta0': 0.001} ...
** FOLD  1
** FOLD  2
** FOLD  3
** FOLD  4
** FOLD  5
** FOLD  6
** FOLD  7
** FOLD  8
** FOLD  9
** FOLD  10
*** Tuning Case  {'num_dims': 3, 'eta0': 0.01} ...
** FOLD  1
** FOLD  2
** FOLD  3
** FOLD  4
...

>>> # Now we can compare the results obtained for each case.
>>> results
                                     3-NN      5-NN  final_expectance
{'num_dims': 3, 'eta0': 0.001}  0.963333  0.970000          0.890105
{'num_dims': 3, 'eta0': 0.01}   0.966667  0.963333          0.916240
{'num_dims': 3, 'eta0': 0.1}    0.970000  0.963333          0.935243
{'num_dims': 4, 'eta0': 0.001}  0.956667  0.963333          0.897238
{'num_dims': 4, 'eta0': 0.01}   0.956667  0.963333          0.922415
{'num_dims': 4, 'eta0': 0.1}    0.960000  0.963333          0.947319

>>> # We can also take the best result (respect to the first metric).
>>> best
({'eta0': 0.1, 'num_dims': 3}, 0.97000000000000008)

>>> # We also obtain the best DML algorithm already constructed to be used.
>>> nca_best.fit(X,y)

>>> # If we want, we can look at the detailed results of cross validation for each␣
↪case.
>>> detailed["{'num_dims': 3, 'eta0': 0.01}"]
              3-NN      5-NN  final_expectance
SPLIT 1   0.966667  0.966667          0.923293
SPLIT 2   0.966667  0.966667          0.922091
SPLIT 3   1.000000  0.966667          0.907416
SPLIT 4   0.966667  0.966667          0.903700
SPLIT 5   0.966667  0.966667          0.915030
SPLIT 6   0.966667  0.966667          0.905189
SPLIT 7   0.966667  0.966667          0.922051
SPLIT 8   0.933333  0.933333          0.933400
```

(continues on next page)

(continued from previous page)

```
SPLIT 9    0.966667  1.000000          0.912236
SPLIT 10   0.966667  0.933333          0.917992
MEAN       0.966667  0.963333          0.916240
STD        0.014907  0.017951          0.008888
```

1.27 Installation

- PyPI latest version: `pip install pyDML`.
- From GitHub: clone or download this repository and run the command `python setup.py install` on the root directory.

1.28 Stats

The distance metric learning algorithms in pyDML are being evaluated in several datasets. The results of these experiments are available in the pyDML-Stats repository. In this webpage, the algorithms are tested with some distance-based classifiers in several different situations.

1.29 References

- Fei Wang and Changshui Zhang. "Feature extraction by maximizing the average neighborhood margin". In: Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on. IEEE. 2007, pages 1-8.
- Kilian Q Weinberger and Lawrence K Saul. "Distance metric learning for large margin nearest neighbor classification". In: Journal of Machine Learning Research 10.Feb (2009), pages 207-244.
- Jacob Goldberger et al. "Neighbourhood components analysis". In: Advances in neural information processing systems. 2005, pages 513-520.
- Thomas Mensink et al. "Metric learning for large scale image classification: Generalizing to new classes at near-zero cost". In: Computer Vision–ECCV 2012. Springer, 2012, pages 488-501.
- Jason V Davis et al. "Information-theoretic metric learning". In: Proceedings of the 24th international conference on Machine learning. ACM. 2007, pages 209-216.
- Bac Nguyen, Carlos Morell and Bernard De Baets. "Supervised distance metric learning through maximization of the Jeffrey divergence". In: Pattern Recognition 64 (2017), pages 215-225.
- Amir Globerson and Sam T Roweis. "Metric learning by collapsing classes". In: Advances in neural information processing systems. 2006, pages 451-458.
- Eric P Xing et al. "Distance metric learning with application to clustering with side-information". In: Advances in neural information processing systems. 2003, pages 521-528.
- Yiming Ying and Peng Li. "Distance metric learning with eigenvalue optimization". In: Journal of Machine Learning Research 13.Jan (2012), pages 1-26.
- Matthieu Guillaumin, Jakob Verbeek and Cordelia Schmid. "Is that you? Metric learning approaches for face identification". In: Computer Vision, 2009 IEEE 12th international conference on. IEEE. 2009, pages 498-505.
- Sebastian Mika et al. "Fisher discriminant analysis with kernels". In: Neural networks for signal processing IX, 1999. Proceedings of the 1999 IEEE signal processing society workshop. Ieee. 1999, pages 41-48.

- Lorenzo Torresani and Kuang-chih Lee. “Large margin component analysis”. In: Advances in neural information processing systems. 2007, pages 1385-1392.
- Masashi Sugiyama “Dimensionality reduction of multimodal labeled data by local fisher discriminant analysis”. In: Journal of Machine Learning Research, 2007, vol 8, May, pages 1027-1061.

1.29.1 Related links

- Scikit-Learn
- Scikit-Learn Nearest Neighbors
- Scikit-Learn Nearest Class Mean
- matplotlib
- numpy
- pandas

Python Module Index

d

Index

S

T

U