sklvq Documentation

Release 0.1.2

Rick van Veen

Feb 16, 2021

GETTING STARTED

Scikit-learning vector quantization (sklvq) is a scikit-learn compatible and expandable implementation of Learning Vector Quantization (LVQ) algorithms. The main purpose is to make it easier to compare results by providing a central point for the implementations of the LVQ algorithms.

Currently the package implements three algorithms from the LVQ family, all based on the generalized learning objective, i.e., Generalized Learning Vector Quantization (GLVQ), Generalized Matrix LVQ (GMLVQ) and Local Generalized Matrix LVQ (LGMLVQ).

The package provides a number of activation, discriminant, distance, and solver methods. Please see the Getting started, Documentation, and Tutorial - Examples sections on the left side.

CHAPTER

ONE

INSTALL AND CONTRIBUTE

This page containst the lists of dependencies and install instructions for two scenarios. Firstly, "General use" for anyone who just whishes to use the algorithms. Secondly, instruction on how to install the package in order to be able to build the docs and run tests.

1.1 General Use

This section contains the dependencies and install instructions for regular usage of the sklvq package. If you wish to contribute to the package please see the *Contribute* section.

1.1.1 Dependencies

The sklvq toolbox requires the following packages to be installed:

- numpy
- scipy
- scikit-learn

1.1.2 Installation

Sklvq can be installed using pip:

```
pip install sklvq
```

Another option is to clone the repository and run the setup.py file. The following (terminal/cmd) commands can be used to clone the repository and install sklvq with all dependencies:

```
git clone https://github.com/rickvanveen/sklvq.git
cd sklvq
pip install .
```

Or install using pip from GitHub directly:

```
pip install -U git+https://github.com/rickvanveen/sklvq.git
```

1.2 Contribute

You can contribute to this code through pull requests on GitHub. Please, make sure that your code is coming with unit tests to ensure full coverage and continuous integration in the API. Follow the instruction below in order to install all necessary dependencies for development.

1.2.1 Dependencies

In addition to the regular dependencies, sklvq requires a number of packages for testing and building the documentation:

Testing:

- pytest
- pytest-cov

Documentation:

- sphinx
- sphinx-gallery
- sphinx_rtd_theme
- numpydoc
- matplotlib

1.2.2 Installation

The package can be cloned using the following commands:

```
git clone https://github.com/rickvanveen/sklvq.git
cd sklvq
```

Using the following addition to the *pip* command, one can install the dependencies automatically:

```
pip install .[tests]
```

or in order to be able to build the documentation:

```
pip install .[docs]
```

or simply by passing them at the same time (note the lack of whitespace):

```
pip install .[tests,docs]
```

1.2.3 Running Tests

Every module contains its own test folder. Where every file is prepended with *test_*. The tests can be run by using the following command when in the sklvq module folder:

```
pytest .
```

1.2.4 Building Docs

The html docs can be build using the following command (in the docs folder):

```
make html
```

This will generate a build folder from which the index.html can be opened locally. Other options are also available see the 'Makefile' in the doc folder.

CHAPTER

TWO

API REFERENCE

If you would like to use sklvq algorithms the most relevant part to look at is the "Predictors and Transformers" section. However, the other sections provide information about accepted parameters their range and default values.

2.1 Predictors and Transformers

LVQBaseClass(distance_type, type] =, ...)	Learning Vector Quantization base class

2.1.1 sklvq.models.LVQBaseClass

class sklvq.models.**LVQBaseClass**(*distance_type: Union[str, type] = 'squared-euclidean'*, *distance_params: Optional[dict] = None*, *valid_distances: Optional[List[str]] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *valid_solvers: Optional[List[str]] = None*, *prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean'*, *prototype_n_per_class: int = 1*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, bool] = True*)

Learning Vector Quantization base class

Abstract class for implementing LVQ models. It provides abstract methods with expected call signatures.

Provides a common interface to the solver and other function that require access to the models. Additionally, it implements a number of functions shared by the currently implemented LVQ variations.

See also:

GLVQ*, *GMLVQ*, *LGMLVQ

__init__(*distance_type: Union[str, type] = 'squared-euclidean'*, *distance_params: Optional[dict] = None*, *valid_distances: Optional[List[str]] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *valid_solvers: Optional[List[str]] = None*, *prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean'*, *prototype_n_per_class: int = 1*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, bool] = True*)

Initialize self. See help(type(self)) for accurate signature.

abstract add_partial_gradient(*gradient*, *partial_gradient*, *i_prototype*) → None

To increase performance, the distance gradient methods (should) return only the relevant values. I.e., the gradient of the prototype i_prototype and potentially other parameters linked to this prototype. This partial

gradient needs to added (overwrite) to the correct parts of the actual gradient and this is what this function should do.

Parameters

gradient [ndarray] Same shape as the `get_variables()` would return.

partial_gradient [ndarray] 1d array containing the partial gradient.

i_prototype [int] The index of the prototype to which the partial gradient was computed.

`decision_function`(*X: numpy.ndarray*)

Evaluates the decision function for the samples in X. Shape for binary class is (n_observations,) with the decision values for the "greater" class. In the multiclass case it returns decision values for each class and therefore has the shape (n_observations, n_classes).

Parameters

X [ndarray] The data.

Returns

decision_values [ndarray] Binary case shape is (n_observations,) and the multiclass case (n_observations, n_classes)

`fit`(*X: numpy.ndarray*, *y: numpy.ndarray*)

Fit function that provides the general implementation of the LVQ algorithms. It checks the data, calls before_fit method, calls the solve method of the solver, and the after_fit method.

Parameters

X [ndarray of shape (number of observations, number of dimensions)]

y [ndarray of size (number of observations)]

Returns

self The trained model

abstract `get_model_params`() → Union[tuple, numpy.ndarray]

Should return a view or tuple of views (in correct shape) of the model's parameters. Implementation depends on specific model as model parameters may differ per model.

Returns

ndarray or tuple View or tuple of views of the model's parameters.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

`get_prototypes`() → numpy.ndarray

Return a view into `self._variables` of the the shape of the prototypes (n_prototypes, n_features). At the moment only consistency function, does not actually create the shape and only works after `self.prototypes_` has been set.

Returns

prototypes [ndarray of shape (n_prototypes, n_features)] View into `self._variables` with shape specified above.

get_variables() → numpy.ndarray

Returns the `self._variables` array that owns the memory allocated for the model parameters.

Returns

_variables [ndarray] returns the model's _variables array.

abstract mul_step_size(*step_size: Union[float, numpy.ndarray]*, *gradient: numpy.ndarray*) → None

Should multiply the provided gradient with the provided step size and overwrite the values in `gradient`. Depending on the `step_size` being a float or array different step sizes are used for different model parameters (which also depends on the model if there are more then only prototypes)

Parameters

step_size [float or ndarray] The scalar or list of values containing the step sizes.

gradient [ndarray] Same shape as the `get_variables()` would return.

abstract normalize_variables(*var_buffer: numpy.ndarray*) → None

Should modify the var_buffer as if it was the variables array provided by `get_variables()`.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray or tuple Same shape and size as input, but normalized. How to normalize depends on model implementation.

predict(*X: numpy.ndarray*)

Predict function

The decision is made for the label of the prototype with the minimum decision value, as provided by the `decision_function()`.

Parameters

X [ndarray] The data.

Returns

ndarray of shape (n_observations) Returns the predicted labels.

predict_proba(*X: numpy.ndarray*)

Parameters

X [ndarray] The data.

Returns

confidence_scores [ndarray of shape (n_observations, n_classes)]

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

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

abstract set_model_params(*new_model_params: Union[tuple, numpy.ndarray]*)

Should modify the `self._variables` array. Accepts the new_model_params in the shape of the model's parameters, e.g., prototypes or (prototypes, relevance_matrix).

Always needs to be all the model parameters, can not be used for partial updates.

Parameters

new_model_params [ndarray or tuple] Array or tuple of arrays of the new model's parameters.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

set_prototypes(*new_prototypes: numpy.ndarray*) → None

Accepts a new_prototypes array with the same shape as `self.prototypes_` and overwrites the `self._variables` array by copying the values of the new_prototypes.

Parameters

new_prototypes [ndarray of shape (n_prototypes, n_features)] The new prototypes the model should store.

set_variables(*new_variables: numpy.ndarray*) → None

Modifies the `self._variables` by copying the values of `new_variables` into the memory of `self._variables`.

Parameters

new_variables [ndarray] 1d numpy array that contains all the model parameters in continuous memory

abstract to_model_params_view(*var_buffer: numpy.ndarray*) → Union[tuple, numpy.ndarray]

Should return a single view into the var_buffer or a tuple of views. This depends on the model and its parameters.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray or tuple Should return a view or tuple of views of the model parameters in appropriate shapes.

abstract to_prototypes_view(*var_buffer: numpy.ndarray*) → numpy.ndarray
Should return the prototypes from the provided var_buffer. I.e., it selects/views the appropriate part of memory and reshapes it.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray of shape (n_prototypes, n_features) View into the var_buffer.

Examples using `sklvq.models.LVQBaseClass`

- *Generalized LVQ (GLVQ)*
- *Generalized Matrix LVQ (GMLVQ)*
- *Local Generalized Matrix LVQ (LGMLVQ)*
- *Not a Number LVQ (NaNLVQ)*
- *Pipelines*
- *Cross validation*
- *Grid Search*
- *Learning Behaviour*
- *Receiver Operating Characteristic Curve*
- *Activation Functions*
- *Distance Functions*
- *Discriminant Functions*
- *Solvers*

GLVQ(distance_type, type] =, . . .)	Generalized Learning Vector Quantization

2.1.2 sklvq.models.GLVQ

class sklvq.models.**GLVQ**(*distance_type: Union[str, type] = 'squared-euclidean', distance_params: Optional[dict] = None, activation_type: Union[str, type] = 'sigmoid', activation_params: Optional[dict] = None, discriminant_type: Union[str, type] = 'relative-distance', discriminant_params: Optional[dict] = None, solver_type: Union[str, type] = 'steepest-gradient-descent', solver_params: Optional[dict] = None, prototype_init: str = 'class-conditional-mean', prototype_n_per_class: Union[int, numpy.ndarray] = 1, random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None, force_all_finite: Union[str, bool] = True*)
Generalized Learning Vector Quantization

This model uses the *sklvq.objectives.GeneralizedLearningObjective* as its objective function *[1]*.

Parameters

distance_type [{"squared-euclidean", "euclidean"} or Class, default="squared-euclidean"] The distance function. Can be one from the following list or a custom class:

- **"squared-euclidean"** See *sklvq.distances.SquaredEuclidean*
- **"euclidean"** See *sklvq.distances.Euclidean*

distance_params [dict, optional, default=None] Parameters passed to init of distance class.

activation_type [{"identity", "sigmoid", "soft+", "swish"} or Class, default="sigmoid"] The activation function used in the objective function. Can be any of the activation function in the list or custom class.

- **"identity"** See *sklvq.activations.Identity*
- **"sigmoid"** See *sklvq.activations.Sigmoid*
- **"soft+"** See *sklvq.activations.SoftPlus*
- **"swish"** See *sklvq.activations.Swish*

activation_params [dict, default=None] Parameters passed to init of activation function. See the documentation of the activation functions for parameters and defaults.

discriminant_type ["relative-distance" or Class] The discriminant function. Note that different discriminant type may require to rewrite the `decision_function` and `predict_proba` methods.

- **"relative-distance"** See *sklvq.discriminants.RelativeDistance*

discriminant_params [dict, default=None] Parameters passed to init of discriminant callable. See the documentation of the discriminant functions for parameters and defaults.

solver_type [{"sgd", "wgd", "adam", "lbfgs", "bfgs"},] The solver used for optimization

- **"sgd" or "steepest-gradient-descent"** See *sklvq.solvers.SteepestGradientDescent*.
- **"wgd" or "waypoint-gradient-descent"** See *sklvq.solvers.WaypointGradientDescent*.
- **"adam" or "adaptive-moment-estimation"** See *sklvq.solvers.AdaptiveMomentEstimation*.
- **"bfgs" or "broyden-fletcher-goldfarb-shanno"** See *sklvq.solvers.BroydenFletcherGoldfarbShanno*
- **"lbfgs" or "limited-memory-bfgs"** See `skvlq.solvers.LimitedMemoryBfgs`

solver_params [dict, default=None] Parameters passed to init of solvers. See the documentation of the solvers relevant parameters and defaults.

prototype_init: "class-conditional-mean" or ndarray, default="class-conditional-mean" Default will initiate the prototypes to the class conditional mean with a small random offset. Custom numpy array can be passed to change the initial positions of the prototypes.

prototype_n_per_class: int or np.ndarray, optional, default=1 Default will generate single prototype per class. In the case of unequal number of prototypes per class is needed, provide this as np.ndarray. For example, prototype_n_per_class = np.array([1, 6, 3]) this will result in one prototype for the first class, six for the second, and three for the third. Note that the order needs to be the same as the on in the classes_ attribute, which is equal to calling np.unique(labels).

random_state [int, RandomState instance, default=None] Set the random number generation for reproducibility purposes. Used in random offset of prototypes and shuffling of the data in the solvers.

force_all_finite [{True, "allow-nan"}, default=True] Whether to raise an error on np.inf, np.nan, pd.NA in array. The possibilities are:

- True: Force all values of array to be finite.
- "allow-nan": accepts only np.nan and pd.NA values in array. Values cannot be infinite.

References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

Attributes

classes_ [ndarray of shape (n_classes,)] The original and unique labels found in the data.

prototypes_ [ndarray of shape (n_protoypes, n_features)] Positions of the prototypes after fit(X, labels) has been called.

prototypes_labels_ [ndarray of shape (n_prototypes)] Labels for each prototypes. Labels are indexes to `classes_`

__init__ (*distance_type: Union[str, type] = 'squared-euclidean'*, *distance_params: Optional[dict] = None*, *activation_type: Union[str, type] = 'sigmoid'*, *activation_params: Optional[dict] = None*, *discriminant_type: Union[str, type] = 'relative-distance'*, *discriminant_params: Optional[dict] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *prototype_init: str = 'class-conditional-mean'*, *prototype_n_per_class: Union[int, numpy.ndarray] = 1*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, bool] = True*)
Initialize self. See help(type(self)) for accurate signature.

add_partial_gradient (*gradient*, *partial_gradient*, *i_prototype*) → None
Adds the partial gradient to the correct part of the gradient, which depends on `i_prototype`.

Parameters

gradient [ndarray] Same shape as the `get_variables()` would return.

partial_gradient [ndarray] 1d array containing the partial gradient.

i_prototype [int] The index of the prototype to which the partial gradient was computed.

decision_function (*X: numpy.ndarray*)
Evaluates the decision function for the samples in X. Shape for binary class is (n_observations,) with the decision values for the "greater" class. In the multiclass case it returns decision values for each class and therefore has the shape (n_observations, n_classes).

Parameters

X [ndarray] The data.

Returns

decision_values [ndarray] Binary case shape is (n_observations,) and the multiclass case (n_observations, n_classes)

fit (*X: numpy.ndarray*, *y: numpy.ndarray*)

Fit function that provides the general implementation of the LVQ algorithms. It checks the data, calls before_fit method, calls the solve method of the solver, and the after_fit method.

Parameters

X [ndarray of shape (number of observations, number of dimensions)]

y [ndarray of size (number of observations)]

Returns

self The trained model

get_model_params() → numpy.ndarray
Returns a view of all model parameters, which are only the prototypes.

Returns

ndarray Returns a view of the prototypes as ndarray.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

get_prototypes() → numpy.ndarray
Return a view into `self._variables` of the the shape of the prototypes (n_prototypes, n_features). At the moment only consistency function, does not actually create the shape and only works after `self.prototypes_` has been set.

Returns

prototypes [ndarray of shape (n_prototypes, n_features)] View into `self._variables` with shape specified above.

get_variables() → numpy.ndarray
Returns the `self._variables` array that owns the memory allocated for the model parameters.

Returns

_variables [ndarray] returns the model's _variables array.

mul_step_size(*step_size: Union[int, float]*, *gradient: numpy.ndarray*) → None
As GLVQ only has prototypes that are optimized the step_size should be a single float and can just be used to multiply the gradient inplace.

Parameters

step_size [float or ndarray] The scalar or list of values containing the step sizes.

gradient [ndarray] Same shape as the `get_variables()` would return.

normalize_variables(*var_buffer: numpy.ndarray*) → None
Modifies the var_buffer as if it was the variables array provided by `get_variables()`. As variables only contain prototypes it will now contain the normalized prototypes.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray Same shape and size as input, but normalized.

predict (*X: numpy.ndarray*)
Predict function

The decision is made for the label of the prototype with the minimum decision value, as provided by the `decision_function()`.

Parameters

X [ndarray] The data.

Returns

ndarray of shape (n_observations) Returns the predicted labels.

predict_proba (*X: numpy.ndarray*)

Parameters

X [ndarray] The data.

Returns

confidence_scores [ndarray of shape (n_observations, n_classes)]

score (*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

set_model_params (*new_model_params: numpy.ndarray*) → None
Changes the model's internal parameters. Copies the values of model_params into `self.prototypes_` therefor updating the `self.variables_` array.

Parameters

new_model_params [ndarray of shape (n_prototypes, n_features)] In the case the prototypes.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

set_prototypes(*new_prototypes: numpy.ndarray*) → None
Accepts a new_prototypes array with the same shape as `self.prototypes_` and overwrites the `self._variables` array by copying the values of the new_prototypes.

Parameters

new_prototypes [ndarray of shape (n_prototypes, n_features)] The new prototypes the model should store.

set_variables(*new_variables: numpy.ndarray*) → None
Modifies the `self._variables` by copying the values of `new_variables` into the memory of `self._variables`.

Parameters

new_variables [ndarray] 1d numpy array that contains all the model parameters in continuous memory

to_model_params_view(*var_buffer: numpy.ndarray*) → numpy.ndarray
Should create a view of the variables array in prototype shape.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray Returns the prototypes as ndarray.

to_prototypes_view(*var_buffer: numpy.ndarray*) → numpy.ndarray
Returns the prototypes into the provided var_buffer. I.e., it selects/views the appropriate part of memory and reshapes it.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray of shape (n_prototypes, n_features) Prototype view into the var_buffer.

GMLVQ(distance_type, type] =, . . . [, . . .])	Generalized Matrix Learning Vector Quantization

2.1.3 sklvq.models.GMLVQ

class sklvq.models.**GMLVQ**(*distance_type: Union[str, type] = 'adaptive-squared-euclidean', distance_params: Optional[dict] = None, activation_type: Union[str, type] = 'sigmoid', activation_params: Optional[dict] = None, discriminant_type: Union[str, type] = 'relative-distance', discriminant_params: Optional[dict] = None, solver_type: Union[str, type] = 'steepest-gradient-descent', solver_params: Optional[dict] = None, prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean', prototype_n_per_class: Union[int, numpy.ndarray] = 1, relevance_init='identity', relevance_normalization: bool = True, relevance_n_components: Union[str, int] = 'all', relevance_regularization: Union[int, float] = 0, random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None, force_all_finite: Union[str, bool] = True*)

Generalized Matrix Learning Vector Quantization

This model uses the *sklvq.objectives.GeneralizedLearningObjective* as its objective function *[1]*. In addition to learning the positions of the prototypes it learns a relevance matrix that is used in the distance functions *[2]*.

Parameters

distance_type [{"adaptive-squared-euclidean"} or Class, default="squared-euclidean"] Distance function that employs a relevance matrix in its calculation.

- **"adaptive-squared-euclidean"** See *sklvq.distances.AdaptiveSquaredEuclidean*

distance_params [Dict, default=None] Parameters passed to init of distance callable

activation_type [{"identity", "sigmoid", "soft+", "swish"} or Class, default="sigmoid"] Parameters passed to init of activation function. See the documentation of the activation functions for parameters and defaults.

- **"identity"** See *sklvq.activations.Identity*
- **"sigmoid"** See *sklvq.activations.Sigmoid*
- **"soft+"** See *sklvq.activations.SoftPlus*
- **"swish"** See *sklvq.activations.Swish*

activation_params [Dict, default=None] Parameters passed to init of activation function. See the documentation of activation functions for function dependent parameters and defaults.

discriminant_type [{"relative-distance"} or Class, default = "relative-distance"] The discriminant function. Note that different discriminant type may require to rewrite the `decision_function` and `predict_proba` methods.

- **"relative-distance"** See *sklvq.discriminants.RelativeDistance*

discriminant_params [Dict, default=None] Parameters passed to init of discriminant callable. See the documentation of the discriminant functions for parameters and defaults.

solver_type [{"sgd", "wgd", "adam", "lbfgs", "bfgs"},] The solver used for optimization

- **"sgd" or "steepest-gradient-descent"** See *sklvq.solvers.SteepestGradientDescent*.
- **"wgd" or "waypoint-gradient-descent"** See *sklvq.solvers.WaypointGradientDescent*.

- **"adam" or "adaptive-moment-estimation"** See *sklvq.solvers.AdaptiveMomentEstimation*.
- **"bfgs" or "broyden-fletcher-goldfarb-shanno"** Implementation from scipy package.
- **"lbfgs" or "limited-memory-bfgs"** Implementation from scipy package.

solver_params [dict, default=None] Parameters passed to init of solvers. See the documentation of the solvers relevant parameters and defaults.

prototype_init: "class-conditional-mean" or ndarray, default="class-conditional-mean" Default will initiate the prototypes to the class conditional mean with a small random offset. Custom numpy array can be passed to change the initial positions of the prototypes.

prototype_n_per_class: int or np.ndarray, optional, default=1 Default will generate single prototype per class. In the case of unequal number of prototypes per class is needed, provide this as np.ndarray. For example, prototype_n_per_class = np.array([1, 6, 3]) this will result in one prototype for the first class, six for the second, and three for the third. Note that the order needs to be the same as the on in the classes_ attribute, which is equal to calling np.unique(labels).

relevance_init [{"identity", "random"} or np.ndarray, default="identity"] Default will initiate the omega matrices to be the identity matrix. The rank of the matrix can be reduced by setting the `relevance_n_components` attribute *[3]*.

relevance_normalization: bool, optional, default=True Flag to indicate whether to normalize omega, whenever it is updated, such that the trace of the relevance matrix is equal to 1.

relevance_n_components: str {"all"} or int, optional, default="all" For a square relevance matrix use the string "all" (default). For a rectangular relevance matrix use set the number of components explicitly by providing it as an int.

random_state [int, RandomState instance, default=None] Set the random number generation for reproducibility purposes. Used in random offset of prototypes and shuffling of the data in the solvers. Potentially, also used in the random generation of relevance matrix.

force_all_finite [{True, "allow-nan"}, default=True] Whether to raise an error on np.inf, np.nan, pd.NA in array. The possibilities are:

- True: Force all values of array to be finite.
- "allow-nan": accepts only np.nan and pd.NA values in array. Values cannot be infinite.

References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

[2] Schneider, P., Biehl, M., & Hammer, B. (2009). "Adaptive Relevance Matrices in Learning Vector Quantization" Neural Computation, 21(12), 3532–3561, 2009.

[3] Bunte, K., Schneider, P., Hammer, B., Schleif, F.-M., Villmann, T., & Biehl, M. (2012). "Limited Rank Matrix Learning, discriminative dimension reduction and visualization." Neural Networks, 26, 159–173, 2012.

Attributes

classes_ [ndarray of shape (n_classes,)] Class labels for each output.

prototypes_ [ndarray of shape (n_protoypes, n_features)] Positions of the prototypes after `fit(X, labels)` has been called.

prototypes_labels_ [ndarray of shape (n_prototypes)] Labels for each prototypes. Labels are indexes to `classes_`

omega_: ndarray with size depending on initialization, default (n_features, n_features) Omega matrix that was found during training and defines the relevance matrix `lambda_`.

lambda_: ndarray of size (n_features, n_features) The relevance matrix `omega_.T.dot(omega_)`

omega_hat_: ndarray The omega matrix found by the eigenvalue decomposition of the relevance matrix `lambda_`. The eigenvectors (columns of `omega_hat_`) can be used to transform the X *[3]*.

eigenvalues_: ndarray The corresponding eigenvalues to `omega_hat_` found by the eigenvalue decomposition of the relevance matrix `lambda_`

__init__(*distance_type: Union[str, type] = 'adaptive-squared-euclidean'*, *distance_params: Optional[dict] = None*, *activation_type: Union[str, type] = 'sigmoid'*, *activation_params: Optional[dict] = None*, *discriminant_type: Union[str, type] = 'relative-distance'*, *discriminant_params: Optional[dict] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean'*, *prototype_n_per_class: Union[int, numpy.ndarray] = 1*, *relevance_init='identity'*, *relevance_normalization: bool = True*, *relevance_n_components: Union[str, int] = 'all'*, *relevance_regularization: Union[int, float] = 0*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, bool] = True*)

Initialize self. See help(type(self)) for accurate signature.

add_partial_gradient(*gradient*, *partial_gradient*, *i_prototype*) → None

Adds the partial gradient to the correct part of the gradient, which depends on `i_prototype`.

Parameters

gradient [ndarray] Same shape as the `get_variables()` would return.

partial_gradient [ndarray] 1d array containing the partial gradient.

i_prototype [int] The index of the prototype to which the partial gradient was computed.

decision_function(*X: numpy.ndarray*)

Evaluates the decision function for the samples in X. Shape for binary class is (n_observations,) with the decision values for the "greater" class. In the multiclass case it returns decision values for each class and therefore has the shape (n_observations, n_classes).

Parameters

X [ndarray] The data.

Returns

decision_values [ndarray] Binary case shape is (n_observations,) and the multiclass case (n_observations, n_classes)

fit(*X: numpy.ndarray*, *y: numpy.ndarray*)

Fit function that provides the general implementation of the LVQ algorithms. It checks the data, calls before_fit method, calls the solve method of the solver, and the after_fit method.

Parameters

X [ndarray of shape (number of observations, number of dimensions)]

y [ndarray of size (number of observations)]

Returns

self The trained model

fit_transform(*data: numpy.ndarray*, *y: numpy.ndarray*, ***transform_params*) → numpy.ndarray

Parameters

data [ndarray with shape (n_samples, n_features)] Data used for fit and that will be transformed.

y [np.ndarray with length (n_samples)] Labels corresponding to the X samples.

transform_params : Parameters passed to transform function

Returns

The data projected on columns of `omega_hat_` with shape (n_samples, n_columns)

get_model_params() → Tuple[numpy.ndarray, numpy.ndarray]

Returns a tuple of all model parameters. In this case the prototypes and omega matrix.

Returns

ndarray Returns a tuple of views, i.e., the prototypes and omega matrix.

get_omega()

Convenience function to return `self.omega_`

Returns

ndarray, with shape depending on initialization of omega.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

get_prototypes() → numpy.ndarray

Return a view into `self._variables` of the the shape of the prototypes (n_prototypes, n_features). At the moment only consistency function, does not actually create the shape and only works after `self.prototypes_` has been set.

Returns

prototypes [ndarray of shape (n_prototypes, n_features)] View into `self._variables` with shape specified above.

get_variables() → numpy.ndarray

Returns the `self._variables` array that owns the memory allocated for the model parameters.

Returns

_variables [ndarray] returns the model's _variables array.

mul_step_size(*step_sizes: Union[int, float, numpy.ndarray]*, *gradient: numpy.ndarray*) → None

If step sizes is a scalar value just multiplies the gradient with the step size. If it is an array (with same length as number of model parameters) each model parameter is multiplied by its own step size.

Parameters

step_sizes [float or ndarray] The scalar or list of values containing the step sizes.

gradient [ndarray] Same shape as the `get_variables()` would return.

normalize_variables(*var_buffer: numpy.ndarray*) → None
Modifies the var_buffer as if it was the variables array provided by `get_variables()`. Will select, reshape and normalize the correct parts of the variable buffer.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

predict(*X: numpy.ndarray*)
Predict function

The decision is made for the label of the prototype with the minimum decision value, as provided by the `decision_function()`.

Parameters

X [ndarray] The data.

Returns

ndarray of shape (n_observations) Returns the predicted labels.

predict_proba(*X: numpy.ndarray*)

Parameters

X [ndarray] The data.

Returns

confidence_scores [ndarray of shape (n_observations, n_classes)]

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

set_model_params(*new_model_params: Tuple[numpy.ndarray, numpy.ndarray]*)
Changes the model's internal parameters. Copies the values of model_params into `self.prototypes_` and `self.omega_` therefor updating the `self.variables_` array.

Also normalized the relevance matrix if necessary.

Parameters

new_model_params [tuple of ndarrays] Shapes depend on initialization but in the case of a square relevance matrix: tuple((n_prototypes, n_features), (n_features, n_features))

set_omega(*omega*)
Convenience function that makes sure to copy the value to `self.omega_` and not overwrite it.

Parameters

omega [ndarray with same shape as `self.omega_`]

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

set_prototypes(*new_prototypes: numpy.ndarray*) → None
Accepts a new_prototypes array with the same shape as `self.prototypes_` and overwrites the `self._variables` array by copying the values of the new_prototypes.

Parameters

new_prototypes [ndarray of shape (n_prototypes, n_features)] The new prototypes the model should store.

set_variables(*new_variables: numpy.ndarray*) → None
Modifies the `self._variables` by copying the values of `new_variables` into the memory of `self._variables`.

Parameters

new_variables [ndarray] 1d numpy array that contains all the model parameters in continuous memory

to_model_params_view(*var_buffer: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

tuple Returns a tuple with the prototypes and omega matrix as ndarrays.

to_omega(*var_buffer: numpy.ndarray*) → numpy.ndarray
Returns a view (of the shape of the model's omega) into the provided variables buffer of the same size as the model's variables array.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray Shape depending on initialization but in case of a square matrix (n_features, n_features.

to_prototypes_view(*var_buffer: numpy.ndarray*) → numpy.ndarray
Returns a view (of the shape of the model's prototypes) into the provided variables buffer of the same size as the model's variables array.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray of shape (n_prototypes, n_features) Prototype view into the var_buffer.

transform(*X: numpy.ndarray*, *scale: bool = False*) → numpy.ndarray

Parameters

X [np.ndarray with shape (n_samples, n_features)] Data that needs to be transformed

scale [{True, False}, default = False] Controls if the eigenvectors the data is projected on are scaled by the square root of their eigenvalues.

Returns

The data projected on columns of `omega_hat_` with shape (n_samples, n_columns)

LGMLVQ(distance_type, type] =, . . .)	Localized Generalized Matrix Learning Vector Quantization

2.1.4 sklvq.models.LGMLVQ

class sklvq.models.**LGMLVQ**(*distance_type: Union[str, type] = 'local-adaptive-squared-euclidean'*, *distance_params: Optional[dict] = None*, *activation_type: Union[str, type] = 'identity'*, *activation_params: Optional[dict] = None*, *discriminant_type: Union[str, type] = 'relative-distance'*, *discriminant_params: Optional[dict] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean'*, *prototype_n_per_class: [<class 'int'>, <class 'numpy.ndarray'>] = 1*, *relevance_init: Union[str, numpy.ndarray] = 'identity'*, *relevance_normalization: bool = True*, *relevance_n_components: Union[str, int] = 'all'*, *relevance_localization: str = 'prototypes'*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, int] = True*)

Localized Generalized Matrix Learning Vector Quantization

This model uses the *sklvq.objectives.GeneralizedLearningObjective* as its objective function *[1]*. In addition to learning the positions of the prototypes it learns a set of relevance matrices in a localized manner, that are used in the distance functions *[2]*.

Parameters

distance_type ["local-adaptive-squared-euclidean" or Class] Distance function that employs multiple relevance matrix in its calculation. This is controlled by the localization setting.

- **"local-adaptive-squared-euclidean"** See *sklvq.distances.LocalAdaptiveSquaredEuclidean*

distance_params [dict, default=None] Parameters passed to init of distance class.

activation_type [{"identity", "sigmoid", "soft+", "swish"} or Class, default="sigmoid"] The activation function used in the objective function. Can be any of the activation function in the list or custom class.

- **"identity"** See *sklvq.activations.Identity*

- **"sigmoid"** See *sklvq.activations.Sigmoid*
- **"soft+"** See *sklvq.activations.SoftPlus*
- **"swish"** See *sklvq.activations.Swish*

activation_params [Dict, default=None] Parameters passed to init of activation function. See the documentation of the activation functions for parameters and defaults.

discriminant_type ["relative-distance" or Class] The discriminant function. Note that different discriminant type may require to rewrite the `decision_function` and `predict_proba` methods.

- **"relative-distance"** See *sklvq.discriminants.RelativeDistance*

discriminant_params [Dict, default=None] Parameters passed to init of discriminant callable. See the documentation of the discriminant functions for parameters and defaults.

solver_type [{"sgd", "wgd", "adam", "lbfgs", "bfgs"},] The solver used for optimization

- **"sgd" or "steepest-gradient-descent"** See *sklvq.solvers.SteepestGradientDescent*.
- **"wgd" or "waypoint-gradient-descent"** See *sklvq.solvers.WaypointGradientDescent*.
- **"adam" or "adaptive-moment-estimation"** See *sklvq.solvers.AdaptiveMomentEstimation*.
- **"bfgs" or "broyden-fletcher-goldfarb-shanno"** Implementation from scipy package.
- **"lbfgs" or "limited-memory-bfgs"** Implementation from scipy package.

solver_params [dict, default=None] Parameters passed to init of solvers. See the documentation of the solvers relevant parameters and defaults.

prototype_init: "class-conditional-mean" or ndarray, default="class-conditional-mean" Default will initiate the prototypes to the class conditional mean with a small random offset. Custom numpy array can be passed to change the initial positions of the prototypes.

prototype_n_per_class: int or np.ndarray, optional, default=1 Default will generate single prototype per class. In the case of unequal number of prototypes per class is needed, provide this as np.ndarray. For example, prototype_n_per_class = np.array([1, 6, 3]) this will result in one prototype for the first class, six for the second, and three for the third. Note that the order needs to be the same as the on in the classes_ attribute, which is equal to calling np.unique(labels).

relevance_init [{"identity", "random"} or np.ndarray, default="identity"]

Default will initiate the omega matrices to be the identity matrix. The rank of the matrix can be reduced by setting the `relevance_n_components` attribute *[3]*.

relevance_normalization: bool, optional, default=True Flag to indicate whether to normalize omega, whenever it is updated, such that the trace of the relevance matrix is equal to 1.

relevance_n_components: str {"all"} or int, optional, default="all" For a square relevance matrix use the string "all" (default). For a rectangular relevance matrix use set the number of components explicitly by providing it as an int.

relevance_localization: {"prototypes", "class"}, default="prototypes" Setting that controls the localization of the relevance matrices. Either per prototype, where each prototype has its own relevance matrix. Or per class where each class has its own relevance matrix. Note that when one prototype per class is used, changing this setting has no effect.

random_state [int, RandomState instance, default=None] Set the random number generation for reproducibility purposes. Used in random offset of prototypes and shuffling of the data in the solvers. Potentially, also used in the random generation of relevance matrix.

force_all_finite [{True, "allow-nan"}, default=True] Whether to raise an error on np.inf, np.nan, pd.NA in array. The possibilities are:

- True: Force all values of array to be finite.
- "allow-nan": accepts only np.nan and pd.NA values in array. Values cannot be infinite.

References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

[2] Schneider, P., Biehl, M., & Hammer, B. (2009). "Adaptive Relevance Matrices in Learning Vector Quantization" Neural Computation, 21(12), 3532–3561, 2009.

[3] Bunte, K., Schneider, P., Hammer, B., Schleif, F.-M., Villmann, T., & Biehl, M. (2012). "Limited Rank Matrix Learning, discriminative dimension reduction and visualization." Neural Networks, 26, 159–173, 2012.

Attributes

classes_ [ndarray of shape (n_classes,)] Class labels for each output.

prototypes_ [ndarray of shape (n_protoypes, n_features)] Positions of the prototypes after `fit(X, labels)` has been called.

prototypes_labels_ [ndarray of shape (n_prototypes)] Labels for each prototypes. Labels are indexes to `classes_`

omega_: ndarray with size (n_matrices, n_features, n_features) `omega_` matrices that were found during training and define the relevance matrices `lambda_`.

lambda_: ndarray of size (n_matrices, n_features, n_features) The relevance matrices `omega_.T.dot(omega_)` per matrix.

omega_hat_: ndarray The omega matrices found by the eigenvalue decomposition of the relevance matrices `lambda_`. The eigenvectors (columns of `omega_hat_`) can be used to transform the data

[3]. This results in multiple possible transformations, one per relevance matrix.

eigenvalues_: ndarray The corresponding eigenvalues to `omega_hat_` found by the eigenvalue decomposition of the relevance matrices `lambda_`

__init__(*distance_type: Union[str, type] = 'local-adaptive-squared-euclidean'*, *distance_params: Optional[dict] = None*, *activation_type: Union[str, type] = 'identity'*, *activation_params: Optional[dict] = None*, *discriminant_type: Union[str, type] = 'relative-distance'*, *discriminant_params: Optional[dict] = None*, *solver_type: Union[str, type] = 'steepest-gradient-descent'*, *solver_params: Optional[dict] = None*, *prototype_init: Union[str, numpy.ndarray] = 'class-conditional-mean'*, *prototype_n_per_class: [<class 'int'>, <class 'numpy.ndarray'>] = 1*, *relevance_init: Union[str, numpy.ndarray] = 'identity'*, *relevance_normalization: bool = True*, *relevance_n_components: Union[str, int] = 'all'*, *relevance_localization: str = 'prototypes'*, *random_state: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*, *force_all_finite: Union[str, int] = True*)

Initialize self. See help(type(self)) for accurate signature.

add_partial_gradient(*gradient*, *partial_gradient*, *i_prototype*) → None

Adds the partial gradient to the correct part of the gradient, which depends on `i_prototype`.

Parameters

gradient [ndarray] Same shape as the `get_variables()` would return.

partial_gradient [ndarray] 1d array containing the partial gradient.

i_prototype [int] The index of the prototype to which the partial gradient was computed.

`decision_function`(*X: numpy.ndarray*)

Evaluates the decision function for the samples in X. Shape for binary class is (n_observations,) with the decision values for the "greater" class. In the multiclass case it returns decision values for each class and therefore has the shape (n_observations, n_classes).

Parameters

X [ndarray] The data.

Returns

decision_values [ndarray] Binary case shape is (n_observations,) and the multiclass case (n_observations, n_classes)

`fit`(*X: numpy.ndarray*, *y: numpy.ndarray*)

Fit function that provides the general implementation of the LVQ algorithms. It checks the data, calls before_fit method, calls the solve method of the solver, and the after_fit method.

Parameters

X [ndarray of shape (number of observations, number of dimensions)]

y [ndarray of size (number of observations)]

Returns

self The trained model

`fit_transform`(*X: numpy.ndarray*, *y: numpy.ndarray*, ***trans_params*) → numpy.ndarray

Parameters

X [ndarray with shape (n_samples, n_features)] Data used for fit and that will be transformed.

y [np.ndarray with length (n_samples)] Labels corresponding to the X samples.

trans_params : Parameters passed to transform function

Returns

The data projected on columns of `omega_hat_` with shape (n_matrices, n_samples,

n_columns)

`get_model_params`() → Tuple[numpy.ndarray, numpy.ndarray]

Returns a tuple of all model parameters. In this case the prototypes and omega matrix.

Returns

ndarray Returns a tuple of views, i.e., the prototypes and omega matrix.

`get_omega`()

Function to return `self.omega_` (consistency)

Returns

ndarray, with shape depending on initialization of omega.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

get_prototypes() → numpy.ndarray
Return a view into `self._variables` of the the shape of the prototypes (n_prototypes, n_features). At the moment only consistency function, does not actually create the shape and only works after `self.prototypes_` has been set.

Returns

prototypes [ndarray of shape (n_prototypes, n_features)] View into `self._variables` with shape specified above.

get_variables() → numpy.ndarray
Returns the `self._variables` array that owns the memory allocated for the model parameters.

Returns

_variables [ndarray] returns the model's _variables array.

mul_step_size(*step_sizes: Union[int, float, numpy.ndarray]*, *gradient: numpy.ndarray*) → None
If step sizes is a scalar value just multiplies the gradient with the step size. If it is an array (with same length as number of model parameters) each model parameter is multiplied by its own step size.

Parameters

step_sizes [float or ndarray] The scalar or list of values containing the step sizes.

gradient [ndarray] Same shape as the `get_variables()` would return.

normalize_variables(*var_buffer: numpy.ndarray*) → None
Modifies the var_buffer as if it was the variables array provided by `get_variables()`. Will select, reshape and normalize the correct parts of the variable buffer.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

predict(*X: numpy.ndarray*)
Predict function

The decision is made for the label of the prototype with the minimum decision value, as provided by the `decision_function()`.

Parameters

X [ndarray] The data.

Returns

ndarray of shape (n_observations) Returns the predicted labels.

predict_proba(*X: numpy.ndarray*)

Parameters

X [ndarray] The data.

Returns

confidence_scores [ndarray of shape (n_observations, n_classes)]

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

set_model_params(*new_model_params: Tuple[numpy.ndarray, numpy.ndarray]*)
Changes the model's internal parameters. Copies the values of model_params into `self.prototypes_` and `self.omega_` therefore updating the `self.variables_` array.

Parameters

new_model_params [tuple of ndarrays] Shapes depend on initialization but in the case of a square relevance matrix: tuple((n_prototypes, n_features), (n_matrices, n_features, n_features))

set_omega(*omega*)
Convenience function that makes sure to copy the value to `self.omega_` and not overwrite it.

Parameters

omega [ndarray with same shape as `self.omega_`]

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

set_prototypes(*new_prototypes: numpy.ndarray*) → None
Accepts a new_prototypes array with the same shape as `self.prototypes_` and overwrites the `self._variables` array by copying the values of the new_prototypes.

Parameters

new_prototypes [ndarray of shape (n_prototypes, n_features)] The new prototypes the model should store.

set_variables(*new_variables: numpy.ndarray*) → None
Modifies the `self._variables` by copying the values of `new_variables` into the memory of `self._variables`.

Parameters

new_variables [ndarray] 1d numpy array that contains all the model parameters in continuous memory

to_model_params_view(*var_buffer: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

tuple Returns a tuple with the prototypes and omega matrices as ndarrays.

to_omega(*var_buffer: numpy.ndarray*) → numpy.ndarray

Returns a view (of the shape of the model's omega) into the provided variables buffer of the same size as the model's variables array.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray Shape depending on initialization but in case of a square matrix (n_matrices, n_features, n_features).

to_prototypes_view(*var_buffer: numpy.ndarray*) → numpy.ndarray

Returns a view (of the shape of the model's prototypes) into the provided variables buffer of the same size as the model's variables array.

Parameters

var_buffer [ndarray] Array with the same size as the model's variables array as returned by `get_variables()`.

Returns

ndarray of shape (n_prototypes, n_features) Prototype view into the var_buffer.

transform(*X: numpy.ndarray*, *scale: bool = False*, *omega_hat_index: Union[int, List[int]] = 0*) → numpy.ndarray

Parameters

X [np.ndarray with shape (n_samples, n_features)] Data that needs to be transformed

scale [{True, False}, default = False] Controls if the eigenvectors the data is projected on are scaled by the square root of their eigenvalues.

omega_hat_index [int or list] The indices of the omega_hats_ the transformation should be computed for.

Returns

The data projected on columns of `omega_hat_` with shape (n_samples,

n_columns, n_matrices)

2.2 Objective Functions

GeneralizedLearningObjective(...)	Generalized learning objective

2.2.1 sklvq.objectives.GeneralizedLearningObjective

class sklvq.objectives.**GeneralizedLearningObjective**(*activation_type: Union[str, type]*, *activation_params: dict*, *discriminant_type: Union[str, type]*, *discriminant_params: dict*)

Generalized learning objective

Class that holds the generalized learning objective function and its gradient as described in *[1]*.

Parameters

activation_type [{"identity", "sigmoid", "soft-plus", "swish"} or type] If string needs to be one of the indicated options. If not a string needs to be a custom activation class. See *sklvq.activations.ActivationBaseClass*.

activation_params [dict or None] The dictionary with the parameters for the activation function or None if it doesn't require any parameters.

discriminant_type: {"relative-distance"} or type Can only be the relative distance. If not a string it can be a custom class. See *sklvq.discriminants.DiscriminantBaseClass*.

discriminant_params [dict or None] The dictionary with the parameters for the discriminant function or None if it doesn't require any parameters.

Notes

Compatible and used within the following models: *GLVQ*, *GMLVQ*, and *LGMLVQ*.

References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

__call__(*model: LVQBaseClass*, *data: numpy.ndarray*, *labels: numpy.ndarray*) → numpy.ndarray

Computes the generalized learning objective:

$$E = \sum_{i=1}^{N} f(\mu(d_0(\mathbf{x}_i), d_1(\mathbf{x}_i))$$

with $\mu(\cdot)$ the discriminative function, $f(\cdot)$ the activation function, and $d_0(\mathbf{x}_i)$ and $d_1(\mathbf{x}_i)$ the shortest distance to a prototype with a different and the same label respectively.

Parameters

model [LVQBaseClass] The model which can be any LVQBaseClass compatible with this objective function.

data: ndarray with shape (n_samples, n_features) The data.

labels: ndarray with shape (n_samples) The labels of the samples in the data.

Returns

float: The cost

gradient (*model: LVQBaseClass*, *data: numpy.ndarray*, *labels: numpy.ndarray*) → numpy.ndarray

Computes the generalized learning objective's gradient with respect to the prototype with a different label:

$$\frac{\partial E}{\partial \mathbf{w}_0} = \frac{\partial f}{\partial \mu} \frac{\partial \mu}{\partial d_0} \frac{\partial d_0}{\partial \mathbf{w}_0}$$

with $\mathbf{w}_0$ the prototype with a different label than the data and d_0 the distance to that prototype.

$$\frac{\partial E}{\partial \mathbf{w}_1} = \frac{\partial f}{\partial \mu} \frac{\partial \mu}{\partial d_1} \frac{\partial d_1}{\partial \mathbf{w}_1}$$

with $\mathbf{w}_1$ the prototype with the same label as the data and d_1 the distance to that prototype.

Parameters

model [LVQBaseClass] The model which can be any LVQBaseClass compatible with this objective function.

data: ndarray with shape (n_samples, n_features) The data.

labels: ndarray with shape (n_samples) The labels of the samples in the data.

Returns

ndarray with the same shape as the model variables array (depending on the model) The generalized learning objective function's gradient

2.3 Activation Functions

ActivationBaseClass()	Activation base class

2.3.1 sklvq.activations.ActivationBaseClass

class sklvq.activations.**ActivationBaseClass**

Activation base class

Abstract class for implementing activation functions. Provides abstract methods with expected call signatures.

Custom activation function '__init__' should accept any parameters as key-value pairs.

See also:

Identity, ***Sigmoid***, ***SoftPlus***, ***Swish***

abstract __call__ (*x: numpy.ndarray*) → numpy.ndarray

Should implement an activation function

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Should perform an elementwise evaluation of some activation function.

abstract gradient (*x: numpy.ndarray*) → numpy.ndarray
Should implement the activation function's gradient

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Should return the elementwise evaluation of the activation function's gradient.

Examples using `sklvq.activations.ActivationBaseClass`

- *Activation Functions*

Identity()	Identity function

2.3.2 sklvq.activations.Identity

class sklvq.activations.**Identity**
Identity function

Class that holds the identity function and gradient.

See also:

Sigmoid, ***SoftPlus***, ***Swish***

__call__ (*x: numpy.ndarray*) → numpy.ndarray

Implementation of the identity function:

$$f(\mathbf{x}) = \mathbf{x}$$

Parameters

x [ndarray of any shape]

Returns

x [ndarray] Elementwise evaluation of the identity function.

gradient (*x: numpy.ndarray*) → numpy.ndarray

The identity functions's gradient:

$$\frac{\partial f}{\partial \mathbf{x}} = \mathbf{1}$$

Parameters

x [ndarray]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the identity function's gradient.

Sigmoid(beta, float] = 1)	Sigmoid function

2.3.3 sklvq.activations.Sigmoid

class sklvq.activations.**Sigmoid**(*beta: Union[int, float] = 1*)

Sigmoid function

Class that holds the sigmoid function and gradient as discussed in *[1]*

Parameters

beta [int or float, optional, default=1] Positive non-zero value that controls the steepness of the Sigmoid function.

See also:

Identity, ***SoftPlus***, ***Swish***

References

[1] Villmann, T., Ravichandran, J., Villmann, A., Nebel, D., & Kaden, M. (2019). "Activation Functions for Generalized Learning Vector Quantization - A Performance Comparison", 2019.

__call__(*x: numpy.ndarray*) → numpy.ndarray

Computes the sigmoid function:

$$f(\mathbf{x}) = \frac{1}{e^{-\beta \cdot \mathbf{x}} + 1}$$

Parameters

x [ndarray of any shape.]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the sigmoid function.

gradient(*x: numpy.ndarray*) → numpy.ndarray

Computes the sigmoid function's gradient with respect to x:

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{(\beta \cdot e^{\beta \cdot \mathbf{x})}}{(e^{\beta \cdot \mathbf{x}} + 1)^2}$$

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the sigmoid function's gradient.

SoftPlus(beta, float] = 1)	Soft+ function

2.3.4 sklvq.activations.SoftPlus

class sklvq.activations.**SoftPlus**(*beta: Union[int, float] = 1*)

Soft+ function

Class that holds the soft+ function and gradient as discussed in *[1]*

Parameters

beta [int or float, optional, default=1] Positive non-zero value that controls the steepness of the Soft+ function.

See also:

Identity, ***Sigmoid***, ***Swish***

References

[1] Villmann, T., Ravichandran, J., Villmann, A., Nebel, D., & Kaden, M. (2019). "Activation Functions for Generalized Learning Vector Quantization - A Performance Comparison", 2019.

__call__(*x: numpy.ndarray*) → numpy.ndarray

Implements the soft+ function:

$$f(\mathbf{x}) = \ln(1 + e^{\beta \cdot \mathbf{x}})$$

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the soft+ function.

gradient(*x: numpy.ndarray*) → numpy.ndarray

Implements the sigmoid function's gradient:

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\beta \cdot e^{\beta \cdot \mathbf{x}}}{1 + e^{\beta \cdot \mathbf{x}}}$$

Parameters

x [ndarray]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the soft+ function's gradient.

Swish(beta, float] = 1)	Swish function

2.3.5 sklvq.activations.Swish

class sklvq.activations.**Swish**(*beta: Union[int, float] = 1*)

Swish function

Class that holds the swish function and gradient as discussed in *[1]*

Parameters

beta [int, float, default=1] Positive non-zero value that controls the steepness of the Swish function.

See also:

Identity, ***Sigmoid***, ***SoftPlus***

References

[1] Villmann, T., Ravichandran, J., Villmann, A., Nebel, D., & Kaden, M. (2019). "Activation Functions for Generalized Learning Vector Quantization - A Performance Comparison", 2019.

__call__(*x: numpy.ndarray*) → numpy.ndarray

Implements the swish function:

$$f(\mathbf{x}) = \frac{\mathbf{x}}{1 + e^{-\beta \cdot \mathbf{x}}}$$

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the swish function.

gradient(*x: numpy.ndarray*) → numpy.ndarray

Implements the sigmoid function's gradient:

$$\frac{\partial f}{\partial \mathbf{x}} = \beta \cdot f(\mathbf{x}) + (\frac{1}{1 + e^{-\beta \cdot \mathbf{x}}}) \cdot (1 - \beta \cdot f(\mathbf{x}))$$

Parameters

x [ndarray of any shape]

Returns

ndarray of shape (x.shape) Elementwise evaluation of the swish function's gradient.

2.4 Discriminant Functions

DiscriminantBaseClass()	Discriminant base class

2.4.1 sklvq.discriminants.DiscriminantBaseClass

class sklvq.discriminants.**DiscriminantBaseClass**

Discriminant base class

Abstract class for implementing discriminant functions. Provides abstract methods with expected call signatures.

Custom discriminative function '__init__' should accept any parameters as key-value pairs.

See also:

RelativeDistance

abstract __call__(*dist_same: numpy.ndarray*, *dist_diff: numpy.ndarray*) → numpy.ndarray

Should implement a discriminant function

Parameters

dist_same [ndarray of shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with the same label.

dist_diff [ndarray of shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with a different label.

Returns

ndarray [with shape (n_samples, 1)] Should perform a elementwise evaluation of a discriminant function.

abstract gradient(*dist_same: numpy.ndarray*, *dist_diff: numpy.ndarray*, *same_label: bool*) → numpy.ndarray

Should implement the discriminant function's gradient

Parameters

dist_same [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with the same label.

dist_diff [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with a different label.

same_label [bool] Indicates if the gradient with respect to a prototype with the same label (True) or with respect to a prototype with a different label (False) needs to be computed.

Returns

ndarray with shape (n_sampeles, 1) Should perform a elementwise evaluation of a discriminant function's gradient.

Examples using `sklvq.discriminants.DiscriminantBaseClass`

- *Discriminant Functions*

RelativeDistance()	Relative distance function

2.4.2 `sklvq.discriminants`.RelativeDistance

class sklvq.discriminants.**RelativeDistance**

Relative distance function

Class that holds the relative distance function and gradient as described in *[1]*.

References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

__call__(*dist_same: numpy.ndarray*, *dist_diff: numpy.ndarray*) → numpy.ndarray

The relative distance discriminant function for a single sample ($\mathbf{x}$):

$$\mu(\mathbf{x}) = \frac{d(\mathbf{x}, \mathbf{w}_1) - d(\mathbf{x}, \mathbf{w}_0)}{d(\mathbf{x}, \mathbf{w}_1) + d(\mathbf{x}, \mathbf{w}_0)},$$

with $\mathbf{w}_1$ the prototype with the same label and $\mathbf{w}_0$ the prototype with a different label.

Parameters

dist_same [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with the same label.

dist_diff [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with a different label.

Returns

ndarray with shape (n_samples, 1) Evaluation of the relative distance discriminative function.

gradient(*dist_same: numpy.ndarray*, *dist_diff: numpy.ndarray*, *same_label: bool*) → numpy.ndarray

Computes the relative distance discriminant function's gradient.

1. The partial derivative with respect to the closest prototypes with the same label (same_label=True):

$$\frac{\partial \mu}{\partial \mathbf{w}_1} = \frac{2 \cdot d(\mathbf{x}, \mathbf{w}_0))}{(d(\mathbf{x}, \mathbf{w}_1) + d(\mathbf{x}, \mathbf{w}_0))^2}.$$

2. The partial derivative with respect to the closest prototypes with a different label (same_label=False):

$$\frac{\partial \mu}{\partial \mathbf{w}_0} = \frac{-2 \cdot d(\mathbf{x}, \mathbf{w}_1))}{(d(\mathbf{x}, \mathbf{w}_1) + d(\mathbf{x}, \mathbf{w}_0))^2},$$

with $d(\mathbf{x}, \mathbf{w}_1)$ the distance to the prototype with the same label and $d(\mathbf{x}, \mathbf{w}_0)$ the distance to the closest prototype with a different label.

Parameters

dist_same [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with the same label.

dist_diff [ndarray with shape (n_samples, 1), with n_samples >= 1] Shortest distance of n_samples to a prototype with a different label.

same_label [bool] Indicating if the derivative with respect to a prototype with the same label (True) or a different label (False) needs to be calculated.

Returns

ndarray with shape (n_samples, 1) Evaluation of the relative distance function's gradient.

2.5 Distance Functions

DistanceBaseClass()	Distance base class

2.5.1 sklvq.distances.DistanceBaseClass

class sklvq.distances.**DistanceBaseClass**

Distance base class

Abstract class for implementing distance functions. It provides abstract methods with expected call signatures.

Custom distance function '__init__' should accept any parameters as key-value pairs.

See also:

Euclidean, ***SquaredEuclidean***, ***AdaptiveSquaredEuclidean***, ***LocalAdaptiveSquaredEuclidean***

abstract __call__(*data: numpy.ndarray*, *model: LVQBaseClass*) → numpy.ndarray

Should implement a distance function.

Parameters

data [ndarray with shape (n_samples, n_features)] The samples for which the distance to the prototypes of the model need to be computed.

model [LVQBaseClass] Any class extending the LVQBaseClass or depending on the type of distance function is implemented a class that provides the required attributes.

Returns

ndarray with shape (n_samples, n_prototypes) Evaluation of the distance between each sample and prototype of the model.

abstract gradient(*data: numpy.ndarray*, *model: LVQBaseClass*, *i_prototype: int*) → numpy.ndarray

The distance gradient method.

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [LVQBaseClass] Any class extending the LVQBaseClass or depending on the type of distance function is implemented, a class that provides the required attributes.

i_prototype [int] The index of the prototype for which the gradient needs to be computed.

Returns

ndarray with shape (n_samples, n_features) The gradient with respect to the prototype (i_prototype) and every sample in X.

Examples using `sklvq.distances.DistanceBaseClass`

- *Distance Functions*

Euclidean()	Euclidean distance function

2.5.2 sklvq.distances.Euclidean

class sklvq.distances.**Euclidean**

Euclidean distance function

Class that holds the euclidean distance function and its gradient.

Parameters

force_all_finite [{True, False, "allow-nan"}] Parameter to indicate that NaNLVQ distance variant should be used. If true no nans are allowed. If False or "allow-nan" nans are allowed.

See also:

SquaredEuclidean, ***AdaptiveSquaredEuclidean***, ***LocalAdaptiveSquaredEuclidean***

Notes

Compatible with the *GLVQ* algorithm (only).

__call__(*data: numpy.ndarray*, *model: GLVQ*) → numpy.ndarray

Computes the Euclidean distance:

$$d(\mathbf{w}, \mathbf{x}) = \sqrt{(\mathbf{x} - \mathbf{w})^\top (\mathbf{x} - \mathbf{w})},$$

with $\mathbf{w}$ a prototype and $\mathbf{x}$ a sample.

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distances to the prototypes of the model need to be computed.

model [GLVQ] A GLVQ model instance, containing the prototypes.

Returns

ndarray with shape (n_samples, n_prototypes) Evaluation of the distance between each sample and prototype of the model.

gradient(*data: numpy.ndarray*, *model: GLVQ*, *i_prototype: int*) → numpy.ndarray

Computes the gradient of the euclidean distance with respect to a single prototype:

$$\frac{\partial d}{\partial \mathbf{w}_i} = -2 \cdot (\mathbf{x} - \mathbf{w}_i)$$

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [GLVQ] A GLVQ model instance.

i_prototype [int] Index of the prototype to compute the gradient for.

Returns

ndarray with shape (n_samples, n_features) The gradient of the prototype with respect to every sample in the data.

SquaredEuclidean()	Squared Euclidean distance

2.5.3 sklvq.distances.SquaredEuclidean

class sklvq.distances.**SquaredEuclidean**

Squared Euclidean distance

Parameters

force_all_finite [{True, False, "allow-nan"}] Parameter to indicate that NaNLVQ distance variant should be used. If true no nans are allowed. If False or "allow-nan" nans are allowed.

See also:

Euclidean, ***AdaptiveSquaredEuclidean***, ***LocalAdaptiveSquaredEuclidean***

Notes

Compatible with the *GLVQ* algorithm (only).

__call__(*data: numpy.ndarray*, *model: GLVQ*) → numpy.ndarray

Computes the squared Euclidean distance:

$$d(\mathbf{w}, \mathbf{x}) = (\mathbf{x} - \mathbf{w})^\top (\mathbf{x} - \mathbf{w}),$$

with $\mathbf{w}$ a prototype and $\mathbf{x}$ a sample.

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [GLVQ] The GLVQ model instance, containing the prototypes.

Returns

ndarray with shape (n_samples, n_prototypes) Evaluation of the distance between each sample and prototype of the model.

gradient (*data: numpy.ndarray*, *model: GLVQ*, *i_prototype: int*) → numpy.ndarray
Computes the gradient of the squared euclidean distance, with respect to a single prototype:

$$\frac{\partial d}{\partial \mathbf{w}_i} = -2 \cdot (\mathbf{x} - \mathbf{w}_i)$$

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [GLVQ] The GLVQ model instance.

i_prototype [int] Index of the prototype to compute the gradient for.

Returns

gradient [ndarray with shape (n_samples, n_features)] The gradient of the prototype with respect to every sample in the data.

AdaptiveSquaredEuclidean()	Adaptive squared Euclidean distance

2.5.4 sklvq.distances.AdaptiveSquaredEuclidean

class sklvq.distances.**AdaptiveSquaredEuclidean**
Adaptive squared Euclidean distance

Class that holds the adaptive squared Euclidean distance function and its gradient as described in *[1]* and *[2]*.

Parameters

force_all_finite [{True, False, "allow-nan"}] Parameter to indicate that NaNLVQ distance variant should be used. If true no nans are allowed. If False or "allow-nan", nans are allowed.

See also:

Euclidean, ***SquaredEuclidean***, ***LocalAdaptiveSquaredEuclidean***

Notes

Compatible with the *GMLVQ* algorithm (only).

References

[1] Schneider, P. (2010). Advanced methods for prototype-based classification. Groningen.

[2] Schneider, P., Biehl, M., & Hammer, B. (2009). Adaptive Relevance Matrices in Learning Vector Quantization. Neural Computation, 21(12), 3532–3561.

__call__ (*data: numpy.ndarray*, *model: GMLVQ*) → numpy.ndarray
Computes the adaptive squared Euclidean distance:

$$d^{\Lambda}(\mathbf{w}, \mathbf{x}) = (\mathbf{x} - \mathbf{w})^{\top} \Lambda (\mathbf{x} - \mathbf{w})$$

with the relevance matrix $\Lambda = \Omega^\top \Omega$, the prototype $\mathbf{w}$, and sample $\mathbf{x}$.

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [GMLVQ] The model instance, containing the prototypes and relevance matrix.

Returns

ndarray with shape (n_samples, n_prototypes) Evaluation of the distance between each sample in the data and prototype of the model.

gradient (*data: numpy.ndarray*, *model: GMLVQ*, *i_prototype: int*) → numpy.ndarray

Computes the gradient of the adaptive squared euclidean distance function, with respect to a single prototype:

$$\frac{\partial d}{\partial \mathbf{w_i}} = -2\Lambda(\mathbf{x} - \mathbf{w_i}),$$

and the omega matrix (per element):

$$\frac{\partial d}{\partial \Omega_{lm}} = 2 \sum_i (x^i - w^i)\Omega_{li}(x^m - w^m)$$

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [GMLVQ] The model instance, containing the prototypes and relevance matrix.

i_prototype [int] An integer index value of the relevant prototype

Returns

ndarray with shape (n_samples, n_features + n_omega_elements) The gradient of the prototype and omega matrix with respect to each data sample.

LocalAdaptiveSquaredEuclidean()	Local adaptive squared Euclidean distance

2.5.5 sklvq.distances.LocalAdaptiveSquaredEuclidean

class sklvq.distances.**LocalAdaptiveSquaredEuclidean**

Local adaptive squared Euclidean distance

Class that holds the localized adaptive squared Euclidean distance function and its gradient as described in *[1]* and *[2]*.

Parameters

force_all_finite [{True, False, "allow-nan"}] Parameter to indicate that NaNLVQ distance variant should be used. If true no nans are allowed. If False or "allow-nan" nans are allowed.

See also:

Euclidean, ***SquaredEuclidean***, ***AdaptiveSquaredEuclidean***

Notes

Compatible with the *LGMLVQ* algorithm (only).

References

[1] Schneider, P. (2010). Advanced methods for prototype-based classification. Groningen.

[2] Schneider, P., Biehl, M., & Hammer, B. (2009). Adaptive Relevance Matrices in Learning Vector Quantization. Neural Computation, 21(12), 3532–3561.

__call__(*data: numpy.ndarray*, *model: LGMLVQ*) → numpy.ndarray
Computes the local variant of the adaptive squared Euclidean distance:

$$d^{\Lambda}(\mathbf{w}, \mathbf{x}) = (\mathbf{x} - \mathbf{w})^{\top} \Omega_j^{\top} \Omega_j (\mathbf{x} - \mathbf{w})$$

with Ω_j depending on the localization setting of the model and $\Lambda_j = \Omega_j^{\top} \Omega_j$. The localization can be either per prototype or per class, see the documentation of *LGMLVQ*.

Parameters

data [ndarray with shape (n_samples, n_features)] The data for which the distance gradient to the prototypes of the model need to be computed.

model [LGMLVQ] A LGMLVQ model instance, containing the prototypes and relevance matrices.

Returns

ndarray with shape (n_samples, n_prototypes) Evaluation of the distance between each sample in the data and prototype of the model.

gradient(*data: numpy.ndarray*, *model: LGMLVQ*, *i_prototype: int*) → numpy.ndarray
Computes the gradient of the localized adaptive squared euclidean distance function with respect to a specified prototype:

$$\frac{\partial d}{\partial \mathbf{w}_i} = -2\Lambda_j(\mathbf{x} - \mathbf{w}_i)$$

and implicitly coupled omega matrix (per element):

$$\frac{\partial d}{\partial \Omega_{lm}} = 2 \sum_i (x^i - w^i) \Omega_{li} (x^m - w^m)$$

Parameters

data [ndarray with shape (n_samples, n_features)] The X for which the distance gradient to the prototypes of the model need to be computed.

model [LGMLVQ] The LGMLVQ model instance, containing the prototypes and relevance matrices.

i_prototype [int] An integer index value of the relevant prototype

Returns

ndarray with shape (n_samples, n_features + n_omega_elements) The gradient of the prototype and omega matrix with respect to each data sample.

2.6 Solvers

SolverBaseClass(objective)	Solver base class

2.6.1 sklvq.solvers.SolverBaseClass

class sklvq.solvers.**SolverBaseClass**(*objective: sklvq.objectives._base.ObjectiveBaseClass*)

Solver base class

Abstract class for implementing solvers. Provides abstract methods with expected calls signatures.

See also:

SteepestGradientDescent, ***WaypointGradientDescent***, ***AdaptiveMomentEstimation***

BroydenFletcherGoldfarbShanno, ***LimitedMemoryBfgs***

abstract solve(*data: numpy.ndarray*, *labels: numpy.ndarray*, *model: LVQBaseClass*) → None

Solve updates the model it is given and does not return anything.

Parameters

data [ndarray of shape (number of observations, number of dimensions)] The data.

labels [ndarray of size (number of observations)] The labels of the samples in the data.

model [LVQBaseClass] The initial model that will also hold the final result

Examples using sklvq.solvers.SolverBaseClass

- *Solvers*

SteepestGradientDescent(objective, max_runs, ...)	Steepest gradient descent (SGD)

2.6.2 sklvq.solvers.SteepestGradientDescent

class sklvq.solvers.**SteepestGradientDescent**(*objective: sklvq.objectives._base.ObjectiveBaseClass*, *max_runs: int = 10*, *batch_size: int = 1*, *step_size: Union[float, numpy.ndarray] = 0.1*, *callback: Optional[callable] = None*)

Steepest gradient descent (SGD)

Implements the steepest gradient descent optimization method. Can perform stochastic, mini-batch and batch gradient descent by changing the batch_size. Implementation is inspired by the description given in *[1]*.

The algorithm performs the following update of the model parameters (θ) per batch. This process is repeated

multiple times (per step) when the `batch_size` (M) is smaller than the total number of samples in the data.

$$\theta = \theta - \eta(t) \cdot \sum_{i}^{M} \nabla e_i(\theta),$$

with $\nabla e_i(\theta)$ the gradient of the objective function with respect to a sample given the current model parameters θ, and $\eta(t)$ the step size at step t, which is changed using a simple annealing function:

$$\eta(t) = \frac{\eta_{init}}{(1 + \frac{t}{t_{max}})},$$

with t_{max} given by the `max_runs` parameter and η_{init} by the `step_size` parameter.

Parameters

objective: ObjectiveBaseClass, required This is set by the algorithm. See *sklvq.models.GLVQ*, *sklvq.models.GMLVQ*, and *sklvq.models.LGMLVQ*.

max_runs: int Maximum number of runs/epochs that will be computed. Should be >= 1. Early stopping can be implemented by providing a `callback` function that returns True when the solver should stop.

batch_size: int Controls the batch size and accepts a value >= 0. The value indicates the number of samples considered to be in the batch. A stochastic gradient descent corresponds with a `batch_size` of 1. For Batch gradient descent 0 can be used to indicate to use all the samples. Any value > 1 < n_samples can be considered as a mini-batch gradient descent.

If batches can not properly be divided in batches with the specified size the last batch might contain less than the specified number of samples.

The data is always shuffled before it is split into batches.

step_size: float or ndarray The step size to control the learning rate of the model parameters. If the same step size should be used for all parameters (e.g., prototypes and omega) then a single float is sufficient. If separate initial step sizes should be used per model parameter then this should be specified by using a numpy array.

callback: callable Callable with signature callable(state). If the callable returns True the solver will stop even if `max_runs` is not reached yet. The state object contains the following:

- **"variables"** Concatenated 1D ndarray of the model's parameters
- **"nit"** The current iteration counter
- **"fun"** The objective cost
- **"step_size"** The current step_size(s)

References

[1] LeKander, M., Biehl, M., & De Vries, H. (2017). "Empirical evaluation of gradient methods for matrix learning vector quantization." 12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization, WSOM 2017.

solve(*data: numpy.ndarray*, *labels: numpy.ndarray*, *model: LVQBaseClass*)
Solve function that gets called by the fit method of the models.

Performs the steps of the steepest gradient descent optimization method.

Parameters

data [ndarray of shape (n_samples, n_features)] The data.

labels [ndarray of size (n_samples)] The labels of the samples in the data.

model [LVQBaseClass] The initial model that will also hold the final result

WaypointGradientDescent(objective, max_runs, ...)	Waypoint gradient descent (WGD)

2.6.3 sklvq.solvers.WaypointGradientDescent

class sklvq.solvers.**WaypointGradientDescent**(*objective: sklvq.objectives._base.ObjectiveBaseClass*, *max_runs: int = 10*, *step_size: Union[float, numpy.ndarray] = 0.1*, *loss: float = 0.6666666666666666*, *gain: float = 1.1*, *k: int = 3*, *callback: Optional[callable] = None*)

Waypoint gradient descent (WGD)

Implements the waypoint average optimization algorithm *[1]*. Implementation and description is inspired by *[2]*.

The algorithm keeps a rolling average of the last `k` model parameters. After `k` steps the algorithms will compare the cost of the average model parameters ($\hat{\theta}$) versus a "regular" update of the model parameters ($\tilde{\theta}$).

$$\tilde{\theta} = \theta_t - \eta \cdot \frac{\nabla E(\theta_t)}{||\nabla E(\theta_t)||}$$

$$\hat{\theta} = \frac{1}{k} \sum_{i=0}^{k-1} \theta_{t_i}$$

If the regular step results in a lower cost ($E(\tilde{\theta}) < E(\hat{\theta})$), the `step_size` is increased by multiplying with the `gain` factor:

$$\theta_{t+1} = \tilde{\theta}$$
$$\eta = gain \cdot \eta.$$

If the average step results in a lower cost (($E(\hat{\theta}) < E(\tilde{\theta})$) the `step_size` is decreased by multiplying with the `loss` factor:

$$\theta_{t+1} = \hat{\theta}$$
$$\eta = loss \cdot \eta.$$

Note that the solver uses the normalized objective gradient to update the model.

Parameters

objective: ObjectiveBaseClass, required

This is set by the algorithm. See *sklvq.models.GLVQ*, *sklvq.models.GMLVQ*, and *sklvq.models.LGMLVQ*.

max_runs: int Maximum number of runs/epochs that will be computed. Should be >= k. Early stopping can be implemented by providing a `callback` function that returns True when the solver should stop.

step_size: float or ndarray The step size to control the learning rate of the model parameters. If the same step size should be used for all parameters (e.g., prototypes and omega) then a single float is sufficient. If separate initial step_sizes should be used per model parameter then this should be specified by using a ndarray.

Whenever the averge update is accepted (has a lower cost) the step sizes are multiplied with the `loss` factor. When the "regular" update is accepted then the step size(s) are multiplied by the `gain` factor.

loss: float Should be a value less than 1. Controls the step size change factor when an average waypoint step is accepted.

gain: float Should be a value greater than 1. Controls the step size change factor when a regular update step is accepted.

k: int The number of runs used to compute the average waypoint over.

callback: callable Callable with signature callable(state). If the callable returns True the solver will stop even if max_runs is not reached yet. The state object contains the following:

- **"variables"** Concatenated 1D ndarray of the model's parameters
- **"nit"** The current iteration counter.
- **"fun"** The accepted cost.
- **"nfun"** The cost of the regular update step.
- **"tfun"** The cost of the "tentative" update, i.e., the average of the past k updates.
- **"step_size"** The current step_size(s)

References

[1] Papari, G., and Bunte, K., and Biehl, M. (2011) "Waypoint averaging and step size control in learning by gradient descent" Mittweida Workshop on Computational Intelligence (MIWOCI) 2011.

[2] LeKander, M., Biehl, M., & De Vries, H. (2017). "Empirical evaluation of gradient methods for matrix learning vector quantization." 12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization, WSOM 2017.

solve(*data: numpy.ndarray*, *labels: numpy.ndarray*, *model: LVQBaseClass*)
Solve function that gets called by the fit method of the models.

Performs the steps of the waypoint gradient descent optimization method.

Parameters

data [ndarray of shape (n_samples, n_features)] The data.

labels [ndarray of size (n_samples)] The labels of the samples in the data.

model [LVQBaseClass] The initial model that will also hold the final result

AdaptiveMomentEstimation(objective, ...)	Adaptive moment estimation (ADAM)

2.6.4 sklvq.solvers.AdaptiveMomentEstimation

class sklvq.solvers.**AdaptiveMomentEstimation**(*objective: sklvq.objectives._base.ObjectiveBaseClass*, *max_runs: int = 10*, *beta1: float = 0.9*, *beta2: float = 0.999*, *step_size: float = 0.001*, *epsilon: float = 0.0001*, *callback: Optional[callable] = None*)

Adaptive moment estimation (ADAM)

Implementation and description inspired by *[1]*.

Adam maintains two moving averages of the gradient (m, v), which get updated for every sample at each epoch/run until the maximum runs (`max_runs`) has been reached:

$$\mathbf{m} = \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \nabla e_i(\theta)$$
$$\mathbf{v} = \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot [\nabla e_i(\theta)]^{\circ 2}.$$

Since m and v are initialized to zero vectors, they are biased towards zero. To counteract this, unbiased estimates $\hat{m}$ and $\hat{v}$ are computed:

$$\hat{\mathbf{m}} = \mathbf{m}/(1 - \beta_1^p)$$
$$\hat{\mathbf{v}} = \mathbf{v}/(1 - \beta_2^p),$$

where p is initially 0, but afterwards it's increased by 1 each time before selecting a new random sample. The unbiased estimates of the average gradient are then used for the update step:

$$\theta = \theta - \eta \cdot \hat{\mathbf{m}} \odot \hat{\mathbf{v}}^{\circ \frac{1}{2}},$$

with η the `step_size`. Additionally, `beta1`, and `beta2`, can be chosen by the user.

Note that $\odot$ denotes the elementwise (Hadamard) product and $\mathbf{x}^{\circ y}$ the elementwise power operation.

Parameters

objective: ObjectiveBaseClass, required This is/should be set by the algorithm.

max_runs: int Number of runs over all the X. Should be >= 1

beta1: float Controls the decay rate of the moving average of the gradient. Should be < 1.0 and > 0.

beta2: float Controls the decay rate of the moving average of the squared gradient. Should be < 1.0 and > 0.

step_size: float The step size to control the learning rate.

epsilon: float Small value to overcome zero division

callback: callable Callable with signature callable(state). If the callable returns True the solver will stop (early). The state object contains the following information:

- **"variables"** Concatenated 1D ndarray of the model's parameters
- **"nit"** The current iteration counter
- **"fun"** The objective cost
- **"m_hat"** Unbiased moving average of the gradient
- **"v_hat"** Unbiased moving average of the Hadamard squared gradient

References

[1] LeKander, M., Biehl, M., & De Vries, H. (2017). "Empirical evaluation of gradient methods for matrix learning vector quantization." 12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization, WSOM 2017.

solve(*data: numpy.ndarray*, *labels: numpy.ndarray*, *model: LVQBaseClass*)

Parameters

data [ndarray of shape (number of observations, number of dimensions)]

labels [ndarray of size (number of observations)]

model [LVQBaseClass] The initial model that will be changed and holds the results at the end

BroydenFletcherGoldfarbShanno(objective, ...)	Broyden Fletcher Goldfarb Shanno (BFGS)

2.6.5 sklvq.solvers.BroydenFletcherGoldfarbShanno

class sklvq.solvers.**BroydenFletcherGoldfarbShanno**(*objective: sklvq.objectives._base.ObjectiveBaseClass*, ***kwargs*)

Broyden Fletcher Goldfarb Shanno (BFGS)

See the documentation of scipy for a complete parameter list and description.

Parameters

jac: None Is set automatically to objective gradient method. However, if no gradient function is available, e.g., for a custom distance function, then jac can be set to None.

callback: callable Differently from the non-scipy solvers the signature is callback(xk) with xk the current set of variables, which are the model parameters flattened to one 1D array.

LimitedMemoryBfgs(objective, **kwargs)	Limited memory variant of BFGS (L-BFGS)

2.6.6 sklvq.solvers.LimitedMemoryBfgs

class sklvq.solvers.**LimitedMemoryBfgs**(*objective: sklvq.objectives._base.ObjectiveBaseClass*, ***kwargs*)

Limited memory variant of BFGS (L-BFGS)

See the documentation of scipy for the parameter list and description.

Parameters

jac: None Is set automatically to objective gradient method. However, if no gradient function is available, e.g., for a custom distance function, then jac can be set to None.

callback: callable Differently from the non-scipy solvers the signature is callback(xk) with xk the current set of variables, which are the model parameters flattened to one 1D array.

CHAPTER

THREE

BASIC USAGE

Examples of how to fit, predict, and transform (when applicable) the data with each of the LVQ algorithms.

3.1 Generalized LVQ (GLVQ)

Example of how to fit the GLVQ *[1]* algorithm on the classic iris dataset.

```
import matplotlib
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler

from sklvq import GLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

# Contains also the target_names and feature_names, which we will use for the plots.
iris = load_iris()

data = iris.data
labels = iris.target
```

3.1.1 Fitting the Model

Scale the data and create a GLVQ object with, e.g., custom distance function, activation function and solver. See the API reference under documentation for defaults and other possible parameters.

```
# Sklearn's standardscaler to perform z-transform
scaler = StandardScaler()

# Compute (fit) and apply (transform) z-transform
data = scaler.fit_transform(data)

# The creation of the model object used to fit the data to.
model = GLVQ(
    distance_type="squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="steepest-gradient-descent",
```

(continues on next page)

(continued from previous page)

```
    solver_params={"max_runs": 20, "step_size": 0.1},
)
```

The next step is to fit the GLVQ object to the data and use the predict method to make the predictions. Note that this example only works on the training data and therefor does not say anything about the generalizability of the fitted model.

```
# Train the model using the iris dataset
model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# To get a sense of the training performance we could print the classification report.
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.96      0.94      0.95        50
           2       0.94      0.96      0.95        50

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

3.1.2 Extracting the Prototypes

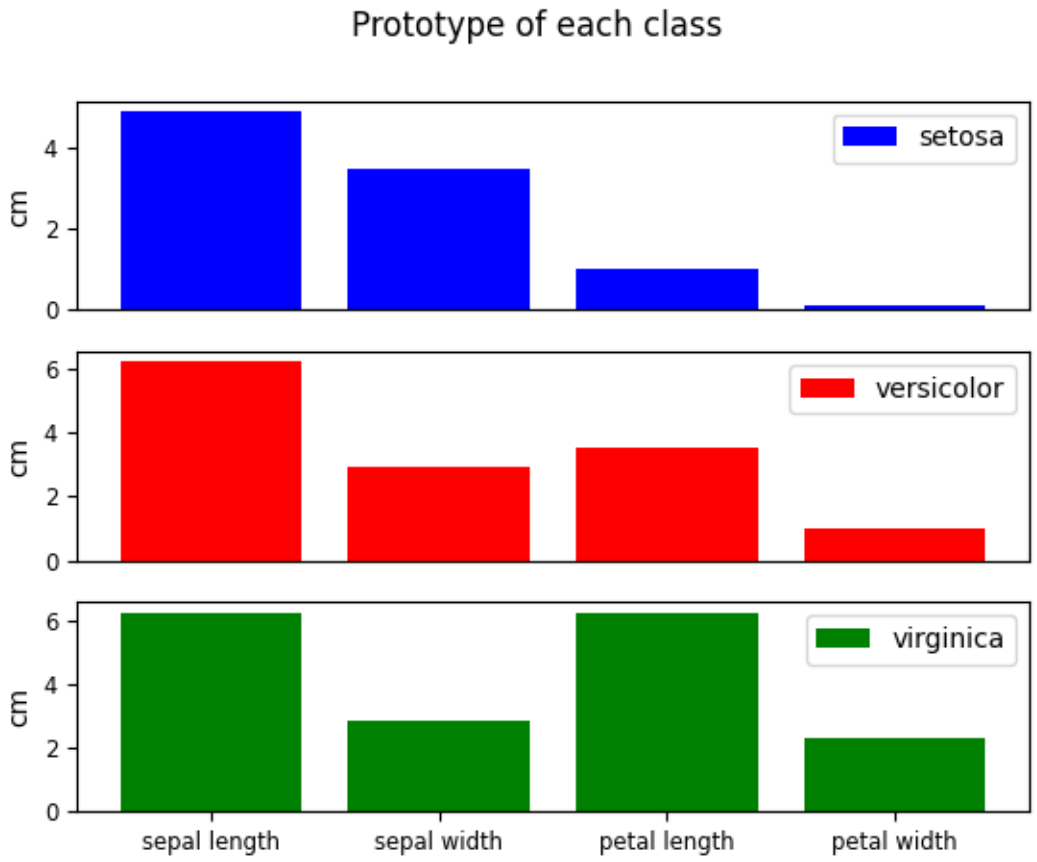
The GLVQ model produces prototypes as representations for the different classes. These prototypes can be accessed and, e.g., plotted for visual inspection. Note that the prototypes of the model are within the z-score space and are transformed back before they are plotted.

```
colors = ["blue", "red", "green"]
num_prototypes = model.prototypes_.shape[0]
num_features = model.prototypes_.shape[1]

fig, ax = plt.subplots(num_prototypes, 1)
fig.suptitle("Prototype of each class")

for i, prototype in enumerate(model.prototypes_):
    # Reverse the z-transform to go back to the original feature space.
    prototype = scaler.inverse_transform(prototype)

    ax[i].bar(
        range(num_features),
        prototype,
        color=colors[i],
        label=iris.target_names[model.prototypes_labels_[i]],
    )
    ax[i].set_xticks(range(num_features))
    if i == (num_prototypes - 1):
        ax[i].set_xticklabels([name[:-5] for name in iris.feature_names])
```

(continues on next page)

(continued from previous page)

```
    else:
        ax[i].set_xticklabels([], visible=False)
        ax[i].tick_params(
            axis="x", which="both", bottom=False, top=False, labelbottom=False
        )
    ax[i].set_ylabel("cm")
    ax[i].legend()
```



3.1.3 References

[1] Sato, A., and Yamada, K. (1996) "Generalized Learning Vector Quantization." Advances in Neural Network Information Processing Systems, 423–429, 1996.

Total running time of the script: (0 minutes 0.637 seconds)

3.2 Generalized Matrix LVQ (GMLVQ)

Example of how to use GMLVQ *[1]* on the classic iris dataset.

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

# Contains also the target_names and feature_names, which we will use for the plots.
iris = load_iris()

data = iris.data
labels = iris.target
feature_names = [name[:-5] for name in iris.feature_names]
```

3.2.1 Fitting the Model

Scale the data and create a GLVQ object with, e.g., custom distance function, activation function and solver. See the API reference under documentation for defaults and other possible parameters.

```
# Sklearn's standardscaler to perform z-transform
scaler = StandardScaler()

# Compute (fit) and apply (transform) z-transform
data = scaler.fit_transform(data)

# The creation of the model object used to fit the data to.
model = GMLVQ(
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={"max_runs": 10, "k": 3, "step_size": np.array([0.1, 0.05])},
    random_state=1428,
)
```

The next step is to fit the GMLVQ object to the data and use the predict method to make the predictions. Note that this example only works on the training data and therefor does not say anything about the generalizability of the fitted model.

```
# Train the model using the scaled data and true labels
model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# To get a sense of the training performance we could print the classification report.
print(classification_report(labels, predicted_labels))
```

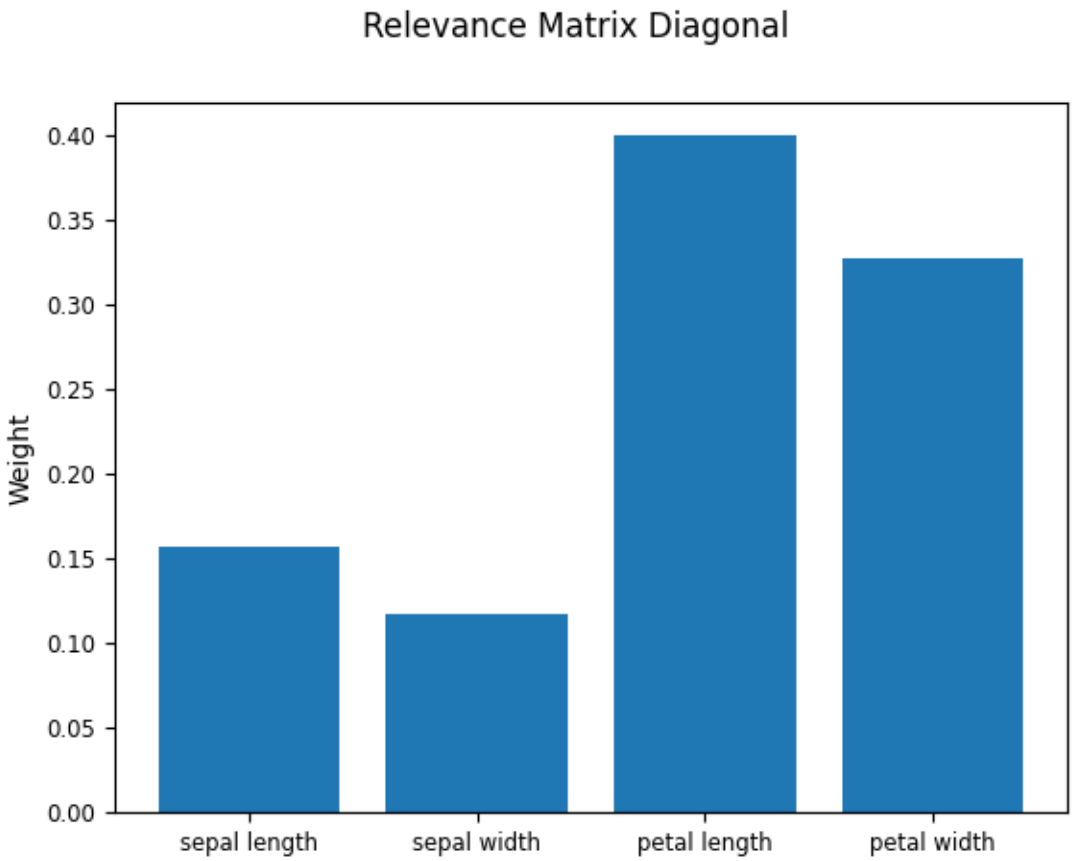
Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.98      0.96      0.97        50
           2       0.96      0.98      0.97        50

    accuracy                           0.98       150
   macro avg       0.98      0.98      0.98       150
weighted avg       0.98      0.98      0.98       150
```

3.2.2 Extracting the Relevance Matrix

In addition to the prototypes (see GLVQ example), GMLVQ learns a matrix *lambda_* which can tell us something about which features are most relevant for the classification.

```
# The relevance matrix is available after fitting the model.
relevance_matrix = model.lambda_

# Plot the diagonal of the relevance matrix
fig, ax = plt.subplots()
fig.suptitle("Relevance Matrix Diagonal")
ax.bar(feature_names, np.diagonal(relevance_matrix))
ax.set_ylabel("Weight")
ax.grid(False)
```



Note that the relevance diagonal adds up to one. The most relevant features for distinguishing between the classes present in the iris dataset seem to be (in decreasing order) the petal length, petal width, sepal length, and sepal width. Although not very interesting for the iris dataset one could use this information to select only the top most relevant features to be used for the classification and thus reducing the dimensionality of the problem.

3.2.3 Transforming the data

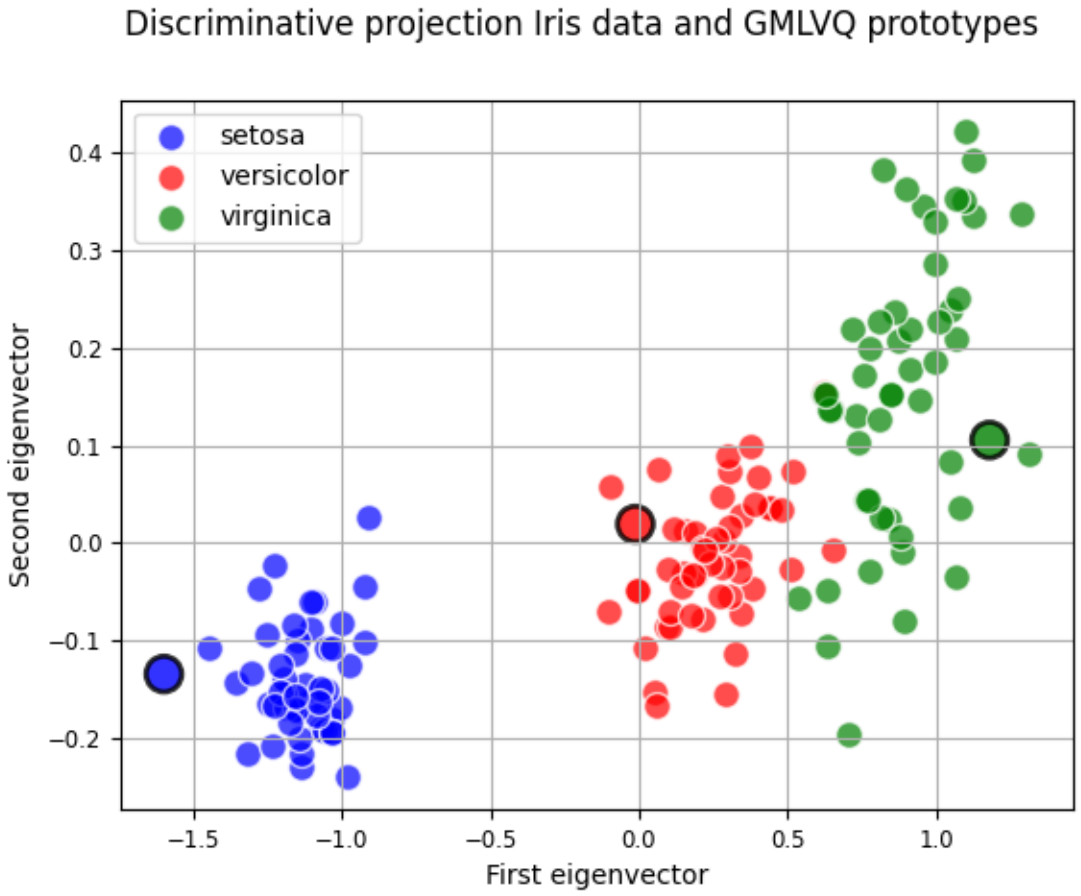
In addition to making predictions GMLVQ can be used to transform the data using the eigenvectors of the relevance matrix.

```
# Transform the data (scaled by square root of eigenvalues "scale = True")
transformed_data = model.transform(data, scale=True)

x_d = transformed_data[:, 0]
y_d = transformed_data[:, 1]

# Transform the model, i.e., the prototypes (scaled by square root of eigenvalues
↪"scale = True")
transformed_model = model.transform(model.prototypes_, scale=True)

x_m = transformed_model[:, 0]
y_m = transformed_model[:, 1]

# Plot
```

(continues on next page)

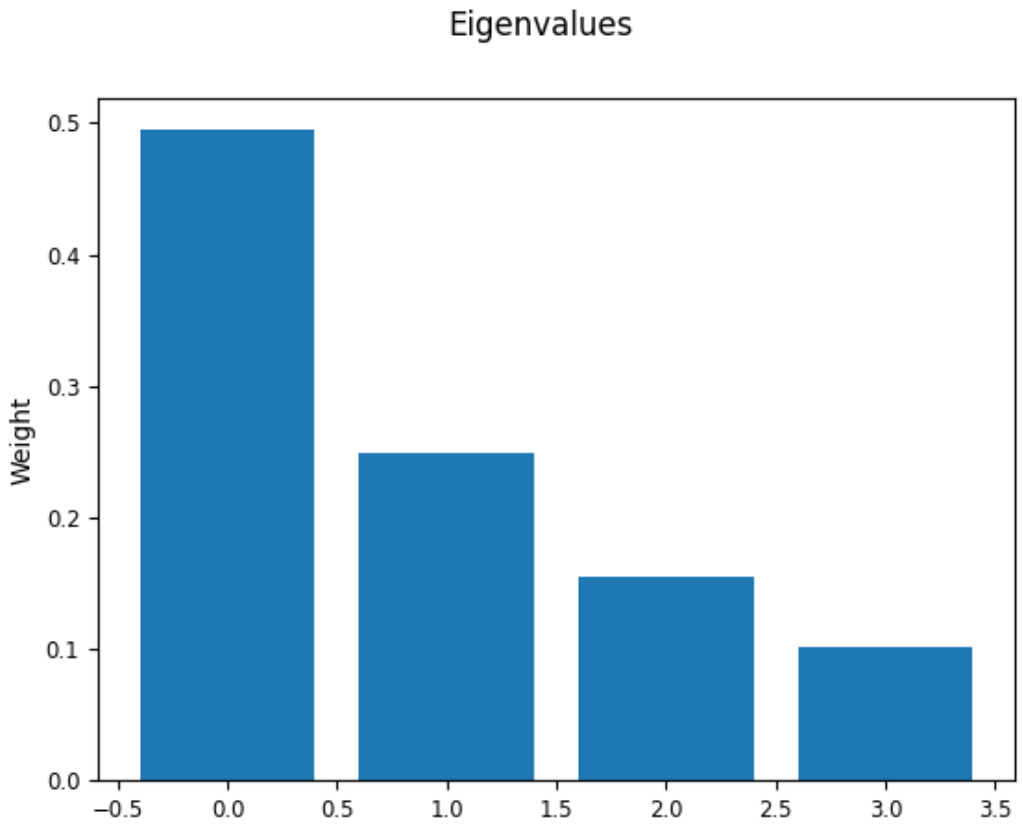
(continued from previous page)

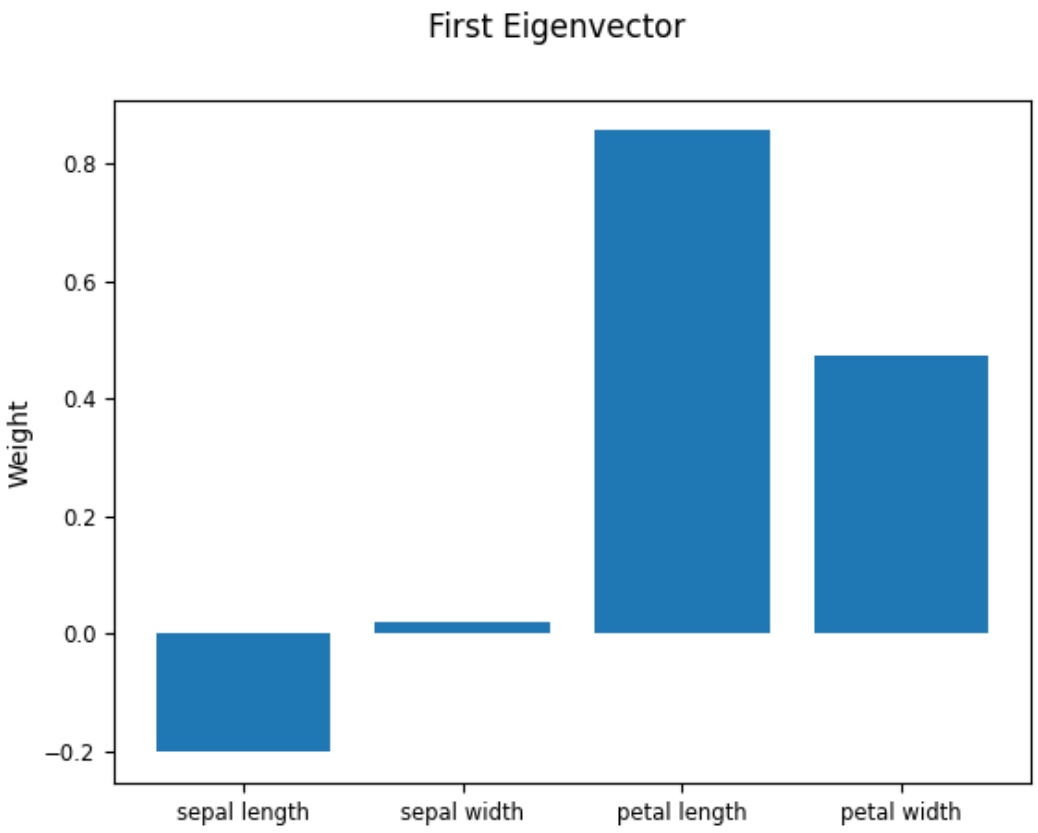
```
fig, ax = plt.subplots()
fig.suptitle("Discriminative projection Iris data and GMLVQ prototypes")
colors = ["blue", "red", "green"]
for i, cls in enumerate(model.classes_):
    ii = cls == labels
    ax.scatter(
        x_d[ii],
        y_d[ii],
        c=colors[i],
        s=100,
        alpha=0.7,
        edgecolors="white",
        label=iris.target_names[model.prototypes_labels_[i]],
    )
ax.scatter(x_m, y_m, c=colors, s=180, alpha=0.8, edgecolors="black", linewidth=2.0)
ax.set_xlabel("First eigenvector")
ax.set_ylabel("Second eigenvector")
ax.legend()
ax.grid(True)
```

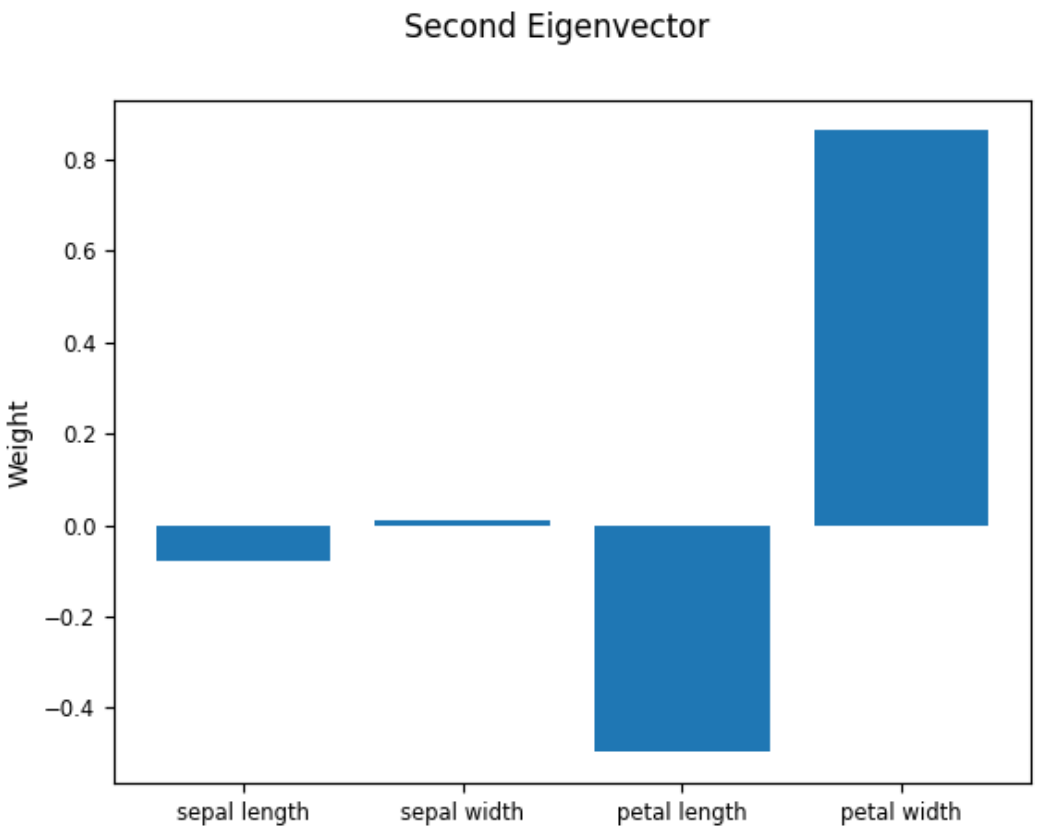


The transformed data and prototypes can be used to visualize the problem in a lower dimension, which is also the space the model would compute the distance. The axis are the directions which are the most discriminating directions (combinations of features). Hence, inspecting the eigenvalues and eigenvectors (axis) themselves can be interesting.

```
# Plot the eigenvalues of the eigenvectors of the relevance matrix.
fig, ax = plt.subplots()
fig.suptitle("Eigenvalues")
ax.bar(range(0, len(model.eigenvalues_)), model.eigenvalues_)
ax.set_ylabel("Weight")
ax.grid(False)

# Plot the first two eigenvectors of the relevance matrix, which  is called `omega_
↪hat`.
fig, ax = plt.subplots()
fig.suptitle("First Eigenvector")
ax.bar(feature_names, model.omega_hat_[:, 0])
ax.set_ylabel("Weight")
ax.grid(False)

fig, ax = plt.subplots()
fig.suptitle("Second Eigenvector")
ax.bar(feature_names, model.omega_hat_[:, 1])
ax.set_ylabel("Weight")
ax.grid(False)
```




First Eigenvector
Weight
0.8
0.6
0.4
0.2
0.0
−0.2
sepal length
sepal width
petal length
petal width



-

In the plots from the eigenvalues and eigenvector we see a similar effects as we could see from just the diagonal of *lambda_*. The two leading (most relevant or discriminating) eigenvectors mostly use the petal length and petal width in their calculation. The diagonal of the relevance matrix can therefor be considered as a summary of the relevances of the features.

3.2.4 References

[1] Schneider, P., Biehl, M., & Hammer, B. (2009). "Adaptive Relevance Matrices in Learning Vector Quantization" Neural Computation, 21(12), 3532–3561, 2009.

Total running time of the script: (0 minutes 0.477 seconds)

3.3 Local Generalized Matrix LVQ (LGMLVQ)

Example of how to use LGMLVQ *[1]* on the classic iris dataset.

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler
```

(continues on next page)

(continued from previous page)

```
from sklvq import LGMLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

# Contains also the target_names and feature_names, which we will use for the plots.
iris = load_iris()

data = iris.data
labels = iris.target
```

3.3.1 Fitting the Model

Scale the data and create a LGMLVQ object with, e.g., custom distance function, activation function and solver. See the API reference under documentation for defaults and other possible parameters.

```
# Sklearn's standardscaler to perform z-transform
scaler = StandardScaler()

# Compute (fit) and apply (transform) z-transform
data = scaler.fit_transform(data)

# The creation of the model object used to fit the data to.
model = LGMLVQ(
    relevance_localization="class",  # Can either be "class" or "prototypes"
    distance_type="local-adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="lbfgs",
)
```

The next step is to fit the LGMLVQ object to the data and use the predict method to make the predictions. Note that this example only works on the training data and therefor does not say anything about the generalizability of the fitted model.

```
# Train the model using the scaled data and true labels
model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# To get a sense of the training performance we could print the classification report.
print(classification_report(labels, predicted_labels))
```

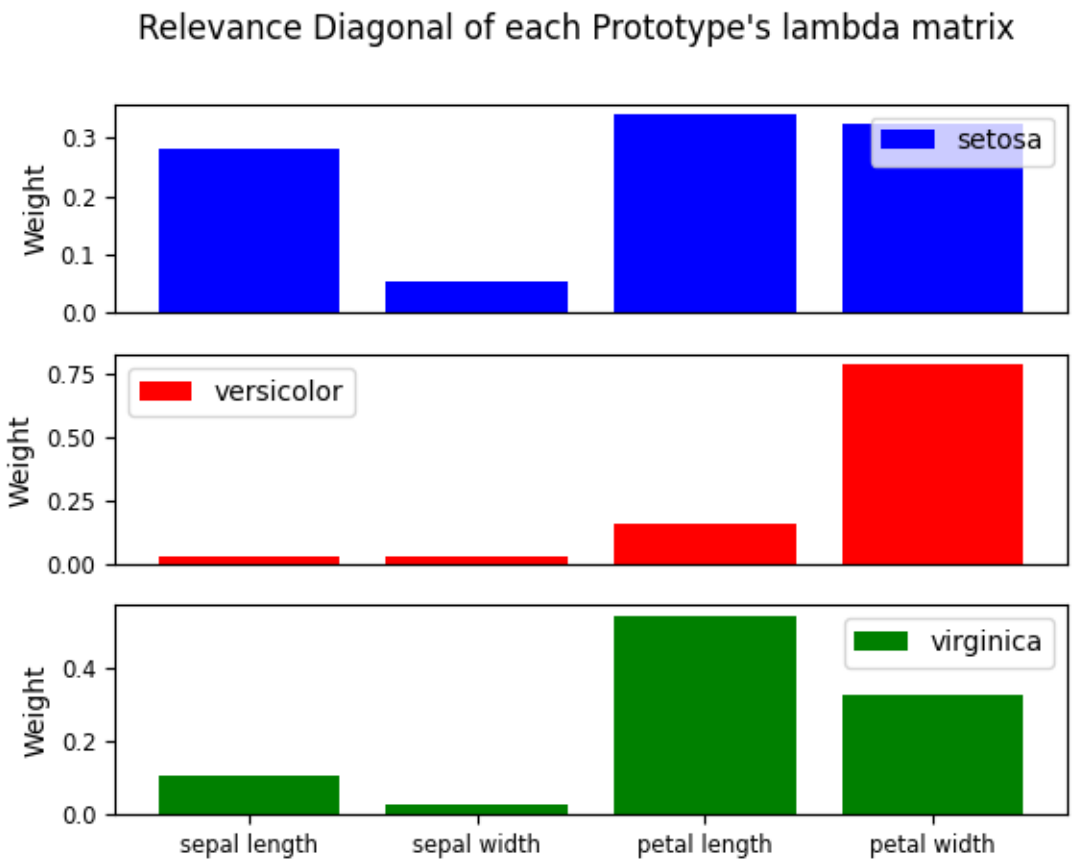
Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.98      1.00      0.99        50
           2       1.00      0.98      0.99        50

    accuracy                           0.99       150
   macro avg       0.99      0.99      0.99       150
weighted avg       0.99      0.99      0.99       150
```

3.3.2 Extracting the Relevance Matrices

In addition to the prototypes (see GLVQ example), LGMLVQ learns a number of matrices *lambda_* which can tell us something about which features are most relevant for the classification per class. The number of relevance matrices is determined by the number of prototypes used per class as well as which localization strategy is used. It can either be a relevance matrix per class (even if there are more prototypes for that class they will share the relevance matrix. Or a relevance matrix per prototype, where each prototype (even if they have the same class) has its own matrix.

```
colors = ["blue", "red", "green"]
num_prototypes = model.prototypes_.shape[0]
num_features = model.prototypes_.shape[1]

fig, ax = plt.subplots(num_prototypes, 1)
fig.suptitle("Relevance Diagonal of each Prototype's lambda matrix")

for i, lambda_ in enumerate(model.lambda_):
    ax[i].bar(
        range(num_features),
        np.diagonal(lambda_),
        color=colors[i],
        label=iris.target_names[model.prototypes_labels_[i]],
    )
    ax[i].set_xticks(range(num_features))
    if i == (num_prototypes - 1):
        ax[i].set_xticklabels([name[:-5] for name in iris.feature_names])
    else:
        ax[i].set_xticklabels([], visible=False)
        ax[i].tick_params(
            axis="x", which="both", bottom=False, top=False, labelbottom=False
        )
    ax[i].set_ylabel("Weight")
    ax[i].legend()
```



Note that each diagonal still adds up to one (See GMLVQ example). However, each diagonal summarizes the importance of the features for its corresponding class versus all other classes.

3.3.3 Transforming the Data

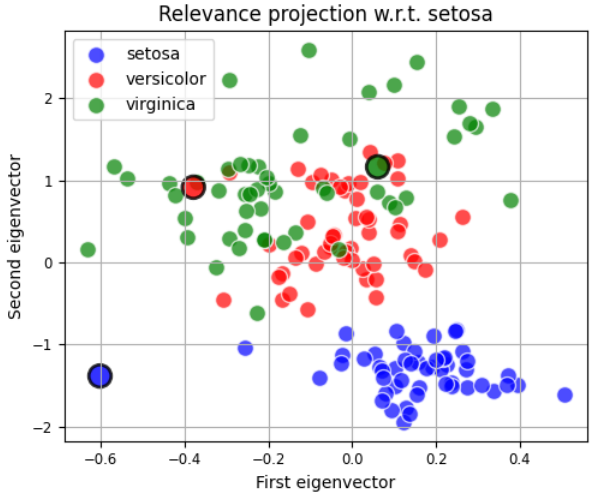
In addition to making predictions LGMLVQ can be used to transform the data using the eigenvectors of the relevance matrices. In contrast to GMLVQ this can be done for each of the matrices separately.

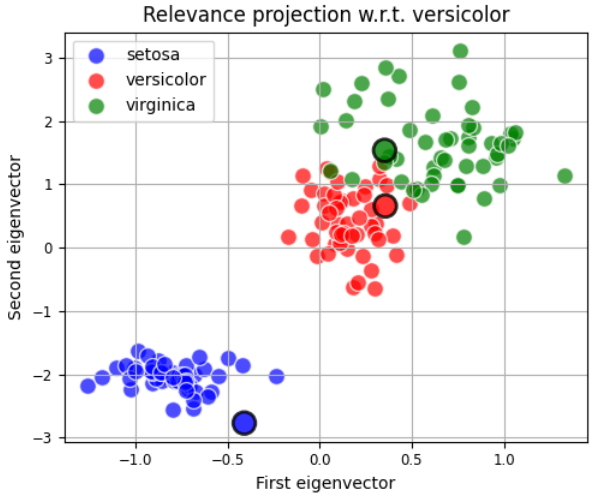
```
# This will return a 3D shape with the 1st and 2nd axes representing the data (n_
↪observations,
# n_eigenvectors). The third axis are the different relevance matrices

t_d = model.transform(data, omega_hat_index=[0, 1, 2])[:, :2, :]
t_d = np.transpose(t_d, axes=(2, 0, 1))  # shape that is easier to work with

t_m = model.transform(model.prototypes_, omega_hat_index=[0, 1, 2])[:, :2, :]
t_m = np.transpose(t_m, axes=(2, 0, 1))  # shape that is easier to work with

fig, ax = plt.subplots(num_prototypes, 1, figsize=(6.4, 14.4))
fig.tight_layout(pad=6.0)

colors = ["blue", "red", "green"]
for i, xy_dm in enumerate(zip(t_d, t_m)):
    xy_d = xy_dm[0]
```

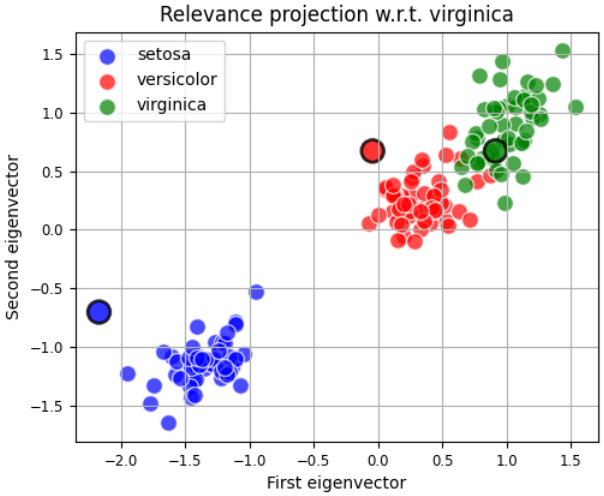
(continues on next page)

(continued from previous page)

```
    xy_m = xy_dm[1]
    for j, cls in enumerate(model.classes_):
        ii = cls == labels
        ax[i].scatter(
            xy_d[ii, 0],
            xy_d[ii, 1],
            c=colors[j],
            s=100,
            alpha=0.7,
            edgecolors="white",
            label=iris.target_names[model.prototypes_labels_[j]],
        )
    ax[i].scatter(
        xy_m[:, 0],
        xy_m[:, 1],
        c=colors,
        s=180,
        alpha=0.8,
        edgecolors="black",
        linewidth=2.0,
    )
    ax[i].title.set_text(
        "Relevance projection w.r.t. {}".format(
            iris.target_names[model.prototypes_labels_[i]]
        )
    )
    ax[i].set_xlabel("First eigenvector")
    ax[i].set_ylabel("Second eigenvector")
    ax[i].legend()
    ax[i].grid(True)
```


Relevance projection w.r.t. setosa
setosa
versicolor
virginica
Second eigenvector
First eigenvector


Relevance projection w.r.t. versicolor
setosa
versicolor
virginica
Second eigenvector
First eigenvector


Relevance projection w.r.t. virginica
setosa
versicolor
virginica
Second eigenvector
First eigenvector

3.3.4 References

[1] Schneider, P., Biehl, M., & Hammer, B. (2009). “Adaptive Relevance Matrices in Learning Vector Quantization” Neural Computation, 21(12), 3532–3561, 2009.

Total running time of the script: (0 minutes 1.400 seconds)

3.4 Not a Number LVQ (NaNLVQ)

NanLVQ *[1]* refers to a extension that can be implemented for various distance functions. It uses the partial distance strategy to ignore any NaN values in the data. Another interpretation would be that it imputes the missing values with those of the prototypes. Hence, the distance will be zero, which results in a zero update for the feature containing the NaN value.

```
import matplotlib
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

iris = load_iris()

data = iris.data
labels = iris.target

# Insert some "random" missing values represented by np.nan
num_missing_values = 50
num_samples, num_dimensions = data.shape

i = np.random.choice(num_samples, num_missing_values, replace=False)
j = np.random.choice(num_dimensions, num_missing_values, replace=True)

data[i, j] = np.nan
```

3.4.1 Fitting the Model

Scale the data and create a GMLVQ object with, e.g., custom distance function, activation function and solver. See the API reference under documentation for defaults and other possible parameters.

```
# Object to perform z-transform
scaler = StandardScaler()

# Compute (fit) and apply (transform) z-transform
data = scaler.fit_transform(data)

# The creation of the model object used to fit the data to.
model = GMLVQ(
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
```

(continues on next page)

(continued from previous page)

```
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={"max_runs": 10, "k": 3, "step_size": np.array([0.1, 0.05])},
    random_state=1428,
    force_all_finite="allow-nan",  # This will make the data  validation  and␣
↪distance function
    # accept and deal with np.nan values.
)
```

The next step is to fit the GMLVQ object to the data and use the predict method to make the predictions. Note that this example only works on the training data and therefor does not say anything about the generalizability of the fitted model.

```
# Train the model using the data and labels
model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# To get a sense of the training performance we could print the classification report.
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.98      0.96      0.97        50
           2       0.96      0.98      0.97        50

    accuracy                           0.98       150
   macro avg       0.98      0.98      0.98       150
weighted avg       0.98      0.98      0.98       150
```

The examples uses GMLVQ but all models and their compatible distance functions support the *force_all_finite* option.

3.4.2 References

[1] Rick van Veen (2016). Analysis of Missing Data Imputation Applied to Heart Failure Data (Master's Thesis, University of Groningen, Groningen, The Netherlands). Retrieved from http://fse.studenttheses.ub.rug.nl/id/eprint/14679

Total running time of the script: (0 minutes 0.062 seconds)

CHAPTER

FOUR

PRE-PROCESSING

Any pre-processing, can be achieved by using the by sklearn's pipelines. Therefore, this section will not discuss the topic in detail but provide a basic example of how one would do this using a model from the sklvq package.

4.1 Pipelines

In these examples GMLVQ is used but the same applies to all the other algorithms. Also the pipelines feature is provided by scikit-learn and we therefore refer to scikit-learn's documentation for more details.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

data, labels = load_iris(return_X_y=True)
```

In previous examples we used a StandardScalar instance to process the data before fitting the model. Sklearn provides a very handy way of creating a connection between the scalar and the model called a pipeline. The pipeline can then be used and will first call the fit method of the standard scaler before the fit of the model. Now the data does not have to be scaled explicitly anymore.

```
# Create a scaler instance
scaler = StandardScaler()

# Create a GMLVQ model (or any other sklearn compatible estimator or other pre-
↪processing method)
model = GMLVQ(
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={"max_runs": 10, "k": 3, "step_size": np.array([0.1, 0.05])},
    random_state=1428,
)

# Link them together into a single object.
pipeline = make_pipeline(scaler, model)

# Fit the data to the pipeline. This will first call the scaler's  fit method before␣
↪passing the
```

(continues on next page)

(continued from previous page)

```
# result to  the model's fit function.
pipeline.fit(data, labels)

# Predict the labels using the trained pipeline. The pipeline will use the
# mean and standard deviation it found when fit was called and applies it to the data.
predicted_labels = pipeline.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.98      0.96      0.97        50
           2       0.96      0.98      0.97        50

    accuracy                           0.98       150
   macro avg       0.98      0.98      0.98       150
weighted avg       0.98      0.98      0.98       150
```

When inspecting the resulting classifier and its prototypes, e.g., in a plot overlaid on a scatter plot of the data, don't forget to apply the scaling to the data:

```
transformed_data = pipeline.transform(data)
```

Total running time of the script: (0 minutes 0.019 seconds)

CHAPTER

FIVE

MODEL SELECTION

This section contains how one could use sklearn's crossvalidation and gridsearch methods in combination with the models provided in the sklvq package.

5.1 Cross validation

In all previous examples we showed the training performance of the models. However, in practice it is much more interesting how well a model performs on unseen data, i.e., the generalizability of the model. We can use crossvalidation for this.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import (
    cross_val_score,
    RepeatedKFold,
)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

data, labels = load_iris(return_X_y=True)
```

Sklearn provides a very handy way of performing cross validation. For this purpose we firstly create a pipeline and initiate a sklearn object that will repeatedly create k folds for us.

```
# Create a scaler instance
scaler = StandardScaler()

# Create a GMLVQ  model instance
model = GMLVQ(
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={"max_runs": 10, "k": 3, "step_size": np.array([0.1, 0.05])},
    random_state=1428,
)

# Link them together (Note this will work as it should in a CV setting, i.e., it's␣
↪fitted to the
# training data and predict is used for the testing data which makes sure the test␣
↪data is
```

(continues on next page)

(continued from previous page)

```
# scaled using the tranformation parameters found during training.
pipeline = make_pipeline(scaler, model)

# Create an object that n_repeat times creates k=10  folds.
repeated_10_fold = RepeatedKFold(n_splits=10, n_repeats=10)

# Call the cross_val_score using all created instances and loaded data. Note it can␣
↪accept
# different and also multiple scoring parameters
accuracy = cross_val_score(
    pipeline, data, labels, cv=repeated_10_fold, scoring="accuracy"
)

# Print the mean and standard deviation of the cross validation testing scores.
print(
    "Accuracy, mean (std): {:.2f} ({:.2f})".format(np.mean(accuracy), np.
↪std(accuracy))
)
```

Out:

```
Accuracy, mean (std): 0.97 (0.05)
```

Total running time of the script: (0 minutes 1.391 seconds)

5.2 Grid Search

Cross validation is not the whole story as it only can tell you the expected performance of a single set of (hyper) parameters. Luckily sklearn also provides a way of trying out multiple settings and return the CV scores for each of them. We can use gridsearch for this.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import GridSearchCV, RepeatedStratifiedKFold
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

data, labels = load_iris(return_X_y=True)
```

We first need to create a pipeline and initialize a parameter grid we want to search.

```
# Create the standard scaler instance
standard_scaler = StandardScaler()

# Create the GMLVQ model instance
model = GMLVQ()

# Link them together by using sklearn's pipeline
pipeline = make_pipeline(standard_scaler, model)

# We want to see the difference in performance of the two following solvers
solvers_types = [
    "steepest-gradient-descent",
```

(continues on next page)

(continued from previous page)

```
    "waypoint-gradient-descent",
]

# Currently, the  sklvq package contains only the following distance function␣
↪compatible with
# GMLVQ. However, see the customization examples for writing your own.
distance_types = ["adaptive-squared-euclidean"]

# Because we are using a pipeline we need to prepend the parameters with the name of␣
↪the
# class of instance we want to provide the parameters for.
param_grid = [
    {
        "gmlvq__solver_type": solvers_types,
        "gmlvq__distance_type": distance_types,
        "gmlvq__activation_type": ["identity"],
    },
    {
        "gmlvq__solver_type": solvers_types,
        "gmlvq__distance_type": distance_types,
        "gmlvq__activation_type": ["sigmoid"],
        "gmlvq__activation_params": [{"beta": beta} for beta in range(1, 4, 1)],
    },
]
# This grid can be read as: for each solver, try each distance type with the identity␣
↪function,
# and the sigmoid activation function for each beta in the range(1, 4, 1)

# Initialize a repeated stratiefiedKFold object
repeated_kfolds = RepeatedStratifiedKFold(n_splits=5, n_repeats=5)

# Initilialize the gridsearch CV instance that will fit the pipeline (standard scaler,
↪ GMLVQ) to
# the data for each of the parameter sets in the grid. Where each fit is a 5 times
# repeated stratified 5 fold cross validation. For each set return the testing␣
↪accuracy.
search = GridSearchCV(
    pipeline,
    param_grid,
    scoring="accuracy",
    cv=repeated_kfolds,
    n_jobs=4,
    return_train_score=False,
    verbose=10,
)

# The gridsearch object can be fitted to the data.
search.fit(data, labels)

# Print the best CV score and parameters.
print("\nBest parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
```

Out:

```
Fitting 25 folds for each of 8 candidates, totalling 200 fits

```

(continues on next page)

(continued from previous page)

```
Best parameter (CV score=0.972):
{'gmlvq__activation_params': {'beta': 1}, 'gmlvq__activation_type': 'sigmoid', 'gmlvq_
↪_distance_type': 'adaptive-squared-euclidean', 'gmlvq__solver_type': 'waypoint-
↪gradient-descent'}
```

When inspecting the resulting classifier and its prototypes, e.g., in a plot overlaid on a scatter plot of the data, don't forget to apply the scaling to the data:

```
transformed_data = search.transform(data)
```

Total running time of the script: (0 minutes 30.358 seconds)

CHAPTER

SIX

PERFORMANCE METRICS

6.1 Learning Behaviour

In these examples GMLVQ is used but the same applies to all the other algorithms. However, not each solver provides the same variables. Additionally, the options "lbfgs" and "bfgs" are implemented in scipy and their callbacks are different from the others. See Scipy's documentation for further information.

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

data, labels = load_iris(return_X_y=True)
```

We create a process logger object and provide it to the solver of the model.

```
class ProcessLogger:
    def __init__(self):
        self.states = np.array([])

    # A callback function has to accept two arguments, i.e., model and state, where␣
↪model is the
    # current model, and state contains a number of the optimizers variables.
    def __call__(self, state):
        self.states = np.append(self.states, state)
        return False  # The callback function can also be used to stop training early,
        # if some condition is met by returning True.


# Initiate the "logger".
logger = ProcessLogger()

scaler = StandardScaler()

model = GMLVQ(
```

(continues on next page)

(continued from previous page)

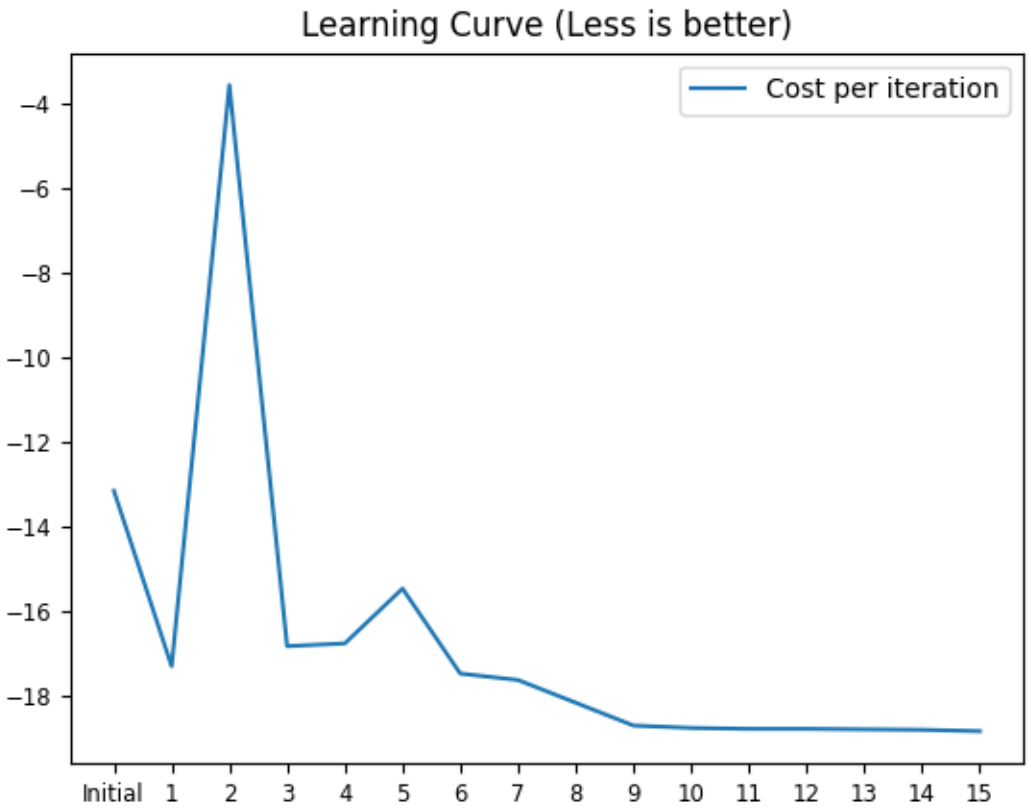
```
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={
        "max_runs": 15,
        "k": 3,
        "step_size": np.array(
            [0.75, 0.85]
        ),  # Note we chose very large step_sizes here to show
        # the usefulness of waypoint averaging.
        "callback": logger,
    },
    random_state=1428,
)

pipeline = make_pipeline(scaler, model)

pipeline.fit(data, labels)

predicted_labels = pipeline.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.98      0.94      0.96        50
           2       0.94      0.98      0.96        50

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

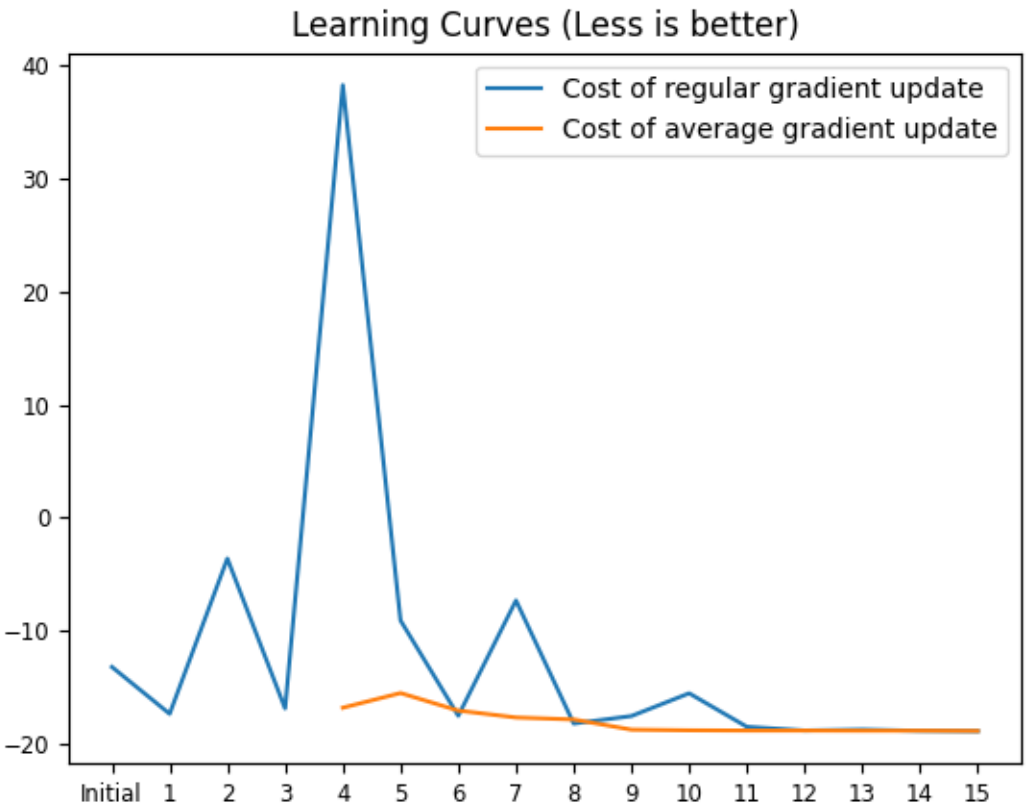
Additionally we can study the cost at each iteration of the solvers progress. Which doesn't look very smooth and even gets worse. This is because of the chosen step_size, which is too large.

```
iteration, fun = zip(*[(state["nit"], state["fun"]) for state in logger.states])

ax = plt.axes()

ax.set_title("Learning Curve (Less is better)")
ax.plot(iteration, fun)
_ = ax.legend(["Cost per iteration"])
```



In the case of waypoint-gradient-descent there is an average cost (tfun) computed over the last k=3 updates and a regular update cost (nfun). Depending on which is less the regular update or the average update is applied.

```
tfun, nfun = zip(*[(state["tfun"], state["nfun"]) for state in logger.states])

ax = plt.axes()

ax.set_title("Learning Curves (Less is better)")
ax.plot(iteration, nfun)
ax.plot(iteration, tfun)
_ = ax.legend(["Cost of regular gradient update", "Cost of average gradient update"])
```



Total running time of the script: (0 minutes 0.244 seconds)

6.2 Receiver Operating Characteristic Curve

Example of plotting the ROC curve for a classification task.

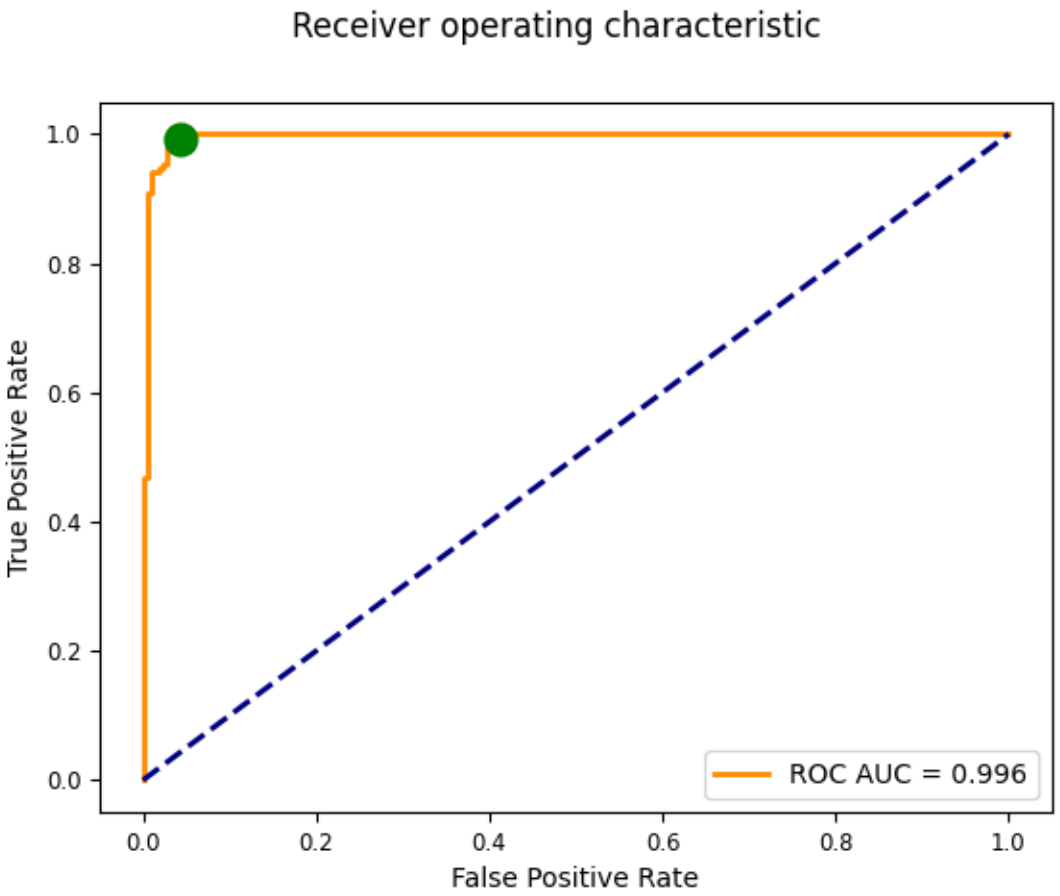
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import roc_curve, roc_auc_score, confusion_matrix
from sklearn.preprocessing import StandardScaler

from sklvq import GMLVQ

matplotlib.rc("xtick", labelsize="small")
matplotlib.rc("ytick", labelsize="small")

data, labels = load_breast_cancer(return_X_y=True)
```

Create a GMLVQ object and pass it a distance function, activation function and solver. See the API reference under documentation for defaults.

```
model = GMLVQ(
    distance_type="adaptive-squared-euclidean",
    activation_type="swish",
    activation_params={"beta": 2},
    solver_type="waypoint-gradient-descent",
    solver_params={"max_runs": 10, "k": 3, "step_size": np.array([0.1, 0.05])},
    random_state=31415,
)
```

Fit the GMLVQ object to the data and plot the roc curve.

```
# Object to perform z-transform
scaler = StandardScaler()

# Compute (fit) and apply (transform) z-transform
data = scaler.fit_transform(data)

# Train the model using the scaled X and true labels
model.fit(data, labels)

# Get the decision values (which are used in predict) instead of the labels. The␣
↪values are with
# respect to the "greater" class, i.e., index 1.
label_score = model.decision_function(data)

# roc_curve expects the y_score to be with respect to the positive class.
fpr, tpr, thresholds = roc_curve(
    y_true=labels, y_score=label_score, pos_label=1, drop_intermediate=True
)
roc_auc = roc_auc_score(y_true=labels, y_score=label_score)

#  Sometimes it is good to know where the Nearest prototype classifier is on this␣
↪curve. This can
#  be computed using the confusion matrix function from sklearn.
tn, fp, fn, tp = confusion_matrix(y_true=labels, y_pred=model.predict(data)).ravel()

# The tpr and fpr of the npc are then given by:
npc_tpr = tp / (tp + fn)
npc_fpr = fp / (fp + tn)

fig, ax = plt.subplots()
fig.suptitle("Receiver operating characteristic ")
# Plot the ROC curve
ax.plot(fpr, tpr, color="darkorange", lw=2, label="ROC AUC = {:.3f}".format(roc_auc))
# Plot the random line
ax.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
# Plot the NPC classifier
ax.plot(npc_fpr, npc_tpr, color="green", marker="o", markersize="12")
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.legend(loc="lower right")
ax.grid(False)
```



Total running time of the script: (0 minutes 0.221 seconds)

CHAPTER

SEVEN

CUSTOMIZATION

The algorithms accept custom activation, discriminant and distance functions, as well as solvers. Any customization with proper testing and documentation will be considered to be included in the sklvq package. Please, create a pull request on github.

Custom models and objectives are also welcome. However, as they greatly impact all currently implemented parts and even might work completely differently there is no easy way to make this similarly expandable as what currently is.

7.1 Activation Functions

```
from typing import Union

import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report

from sklvq import GLVQ
from sklvq.activations import ActivationBaseClass

data, labels = load_iris(return_X_y=True)
```

The sklvq contains already a few activation function. Please see the API reference under Documentation. However, it is fairly easy to create your own. The package works with callable classes and provides a base class for convenience. The base class for the activation functions is sklvq.activations.ActivationBaseClass` and does nothing more then tell you to implement a *__call__()* and *gradient()* method.

```
# This is the implementation of sklvq.activations.Sigmoid with some additional␣
↪comments
class CustomSigmoid(ActivationBaseClass):

    # Activation callables can have a custom init of which the parameters can be␣
↪passed
    # through the `activation_params (Dict)' parameter of the LVQ algorithms. Or the
    # object can just be initialized before hand.
    def __init__(self, beta: Union[int, float] = 1):
        self.beta = beta

    # The activation call function needs to apply the activation elementwise on x.
    def __call__(self, x: np.ndarray) -> np.ndarray:
        return np.asarray(1 / (np.exp(-self.beta * x) + 1))

    # The gradient is the elementwise derivative of the activation function.
```

(continues on next page)

(continued from previous page)

```
    def gradient(self, x: np.ndarray) -> np.ndarray:
        exp = np.exp(self.beta * x)
        return np.asarray((self.beta * exp) / (exp + 1) ** 2)
```

The CustomSigmoid above, accompanied with some tests and documentation, would make a great addition to the sklvq package. However, it can also directly be passed to the algorithm.

```
model = GLVQ(activation_type=CustomSigmoid, activation_params={"beta": 2})

model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.92      0.94      0.93        50
           2       0.94      0.92      0.93        50

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150
```

Total running time of the script: (0 minutes 0.195 seconds)

7.2 Distance Functions

```
from typing import TYPE_CHECKING

import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.metrics.pairwise import pairwise_distances

from sklvq.distances import DistanceBaseClass

if TYPE_CHECKING:
    from sklvq.models import LVQBaseClass

from sklvq import GLVQ

data, labels = load_iris(return_X_y=True)
```

The sklvq contains already a few distance function. Please see the API reference under Documentation. It has a very similar base class to that of the activation functions. However, the structure in which the distance and especially the gradient with respect to the different parameters need to be returned are important. Furthermore not every distance functions works with every algorithm. Below the *sklvq.distances.SquaredEuclidean*, which is suitable for the GLVQ algorithm.

```
class CustomSquaredEuclidean(DistanceBaseClass):

    # The distance implementations use the sklearn pairwise distance function.
    def __init__(self, **other_kwargs):
        self.metric_kwargs = {"metric": "euclidean", "squared": True}

        if other_kwargs is not None:
            self.metric_kwargs.update(other_kwargs)

    # The call function needs to return a matrix with the number of X points on the
    # rows and the columns the distance to the prototypes.
    def __call__(self, data: np.ndarray, model: "LVQBaseClass") -> np.ndarray:
        return pairwise_distances(data, model.prototypes_, **self.metric_kwargs,)

    # The gradient is slightly more difficult as the gradient (with respect to 1
    # prototype) needs to be provided in a vector the size of all the prototypes.
    # Hence, all values are zero except those of the prototype indicated by the index
    # i_prototype. In the case of GMLVQ and LGMVLQ distance functions als the gradient
    # of the omega matrix needs to be returned (in this same vector). See the API
    # reference under Documentation or github for other distance functions and their
    # implementation.
    def gradient(
        self, data: np.ndarray, model: "LVQBaseClass", i_prototype: int
    ) -> np.ndarray:
        prototypes = model.get_model_params()
        (num_samples, num_features) = data.shape

        distance_gradient = np.zeros((num_samples, prototypes.size))

        ip_start = i_prototype * num_features
        ip_end = ip_start + num_features

        distance_gradient[:, ip_start:ip_end] = -2 * (data - prototypes[i_prototype,␣
↪:])

        return distance_gradient
```

The CustomSquaredEuclidean above, accompanied with some tests and documentation, would make a great addition to the sklvq package. However, it can also directly be passed to the algorithm.

```
model = GLVQ(
    distance_type=CustomSquaredEuclidean,
    activation_type="sigmoid",
    activation_params={"beta": 2},
)

model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support
```

(continues on next page)

(continued from previous page)

```
              0       1.00      1.00      1.00        50
              1       0.87      0.92      0.89        50
              2       0.91      0.86      0.89        50

       accuracy                           0.93       150
      macro avg       0.93      0.93      0.93       150
   weighted avg       0.93      0.93      0.93       150
```

Total running time of the script: (0 minutes 0.393 seconds)

7.3 Discriminant Functions

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report

from sklvq import GLVQ
from sklvq.discriminants import DiscriminantBaseClass

data, labels = load_iris(return_X_y=True)
```

The sklvq package contains a single discriminant function and additions are very welcome. Note that they should work with the sklvq.objectives.GeneralizedLearningObjective, i.e., passing additional or different arguments is not possible.

```
# The discriminative function is depended on the objective function. This determines␣
↪the
# parameters of the call and gradient. See sklvq.objective.
↪GeneralizedLearningObjective.
class CustomRelativeDistance(DiscriminantBaseClass):
    def __call__(self, dist_same: np.ndarray, dist_diff: np.ndarray) -> np.ndarray:
        # dist_same = distance to prototype with same label as X.
        # dist_diff = distance to prototype with different label as X.
        return (dist_same - dist_diff) / (dist_same + dist_diff)

    def gradient(
        self, dist_same: np.ndarray, dist_diff: np.ndarray, winner_same: bool
    ) -> np.ndarray:
        # Winner_same is an boolean flag to indicate if the considered prototype has␣
↪the same or
        # a different label compared to the considered X.
        if winner_same:
            return _gradient_same(dist_same, dist_diff)
        return _gradient_diff(dist_same, dist_diff)


# Gradient depends on if the label is the same or different
def _gradient_same(dist_same: np.ndarray, dist_diff: np.ndarray) -> np.ndarray:
    return 2 * dist_diff / (dist_same + dist_diff) ** 2


# Gradient depends on if the label is the same or different
def _gradient_diff(dist_same: np.ndarray, dist_diff: np.ndarray) -> np.ndarray:
    return -2 * dist_same / (dist_same + dist_diff) ** 2
```

The CustomRelativeDistance above, accompanied with some tests and documentation, would make a great addition to the sklvq package. However, it can also directly be passed to the algorithm.

```
model = GLVQ(
    discriminant_type=CustomRelativeDistance,
    distance_type="squared-euclidean",
    activation_type="sigmoid",
    activation_params={"beta": 2},
)

model.fit(data, labels)

# Predict the labels using the trained model
predicted_labels = model.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.92      0.94      0.93        50
           2       0.94      0.92      0.93        50

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150
```

Total running time of the script: (0 minutes 0.192 seconds)

7.4 Solvers

```
from typing import TYPE_CHECKING

import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.utils import shuffle

from sklvq import GLVQ
from sklvq.objectives import ObjectiveBaseClass
from sklvq.solvers import SolverBaseClass
from sklvq.solvers._base import _update_state

if TYPE_CHECKING:
    from sklvq.models import LVQBaseClass

STATE_KEYS = ["variables", "nit", "fun", "step_size"]
```

The sklvq package contains a number of different solvers. Please see the API reference under Documentation for the full list.

```
class CustomSteepestGradientDescent(SolverBaseClass):
    def __init__(
        self,
        # init requires the objective instance to be given when  initialized. It will␣
↪be passed
        # to the (super) solver base class.
        objective: ObjectiveBaseClass,
        max_runs: int = 10,
        batch_size: int = 1,
        step_size: float = 0.1,
        callback: callable = None,
    ):
        super().__init__(objective)
        # In the actual implementation checks can be done to ensure proper values for␣
↪the
        # parameters of the solver (as is done in the actual code).
        self.max_runs = max_runs
        self.batch_size = batch_size
        self.step_size = step_size
        self.callback = callback

    def solve(
        self, data: np.ndarray, labels: np.ndarray, model: "LVQBaseClass",
    ):
        # Calls the callback function is provided with the initial values.
        if self.callback is not None:
            state = _update_state(
                STATE_KEYS,
                variables=np.copy(model.get_variables()),
                nit="Initial",
                fun=self.objective(model, data, labels),
            )
            if self.callback(state):
                return

        batch_size = self.batch_size

        # These checks cannot be done in init because data is not available at that␣
↪moment.
        if batch_size > data.shape[0]:
            raise ValueError("Provided batch_size is invalid.")

        if batch_size <= 0:
            batch_size = data.shape[0]

        for i_run in range(0, self.max_runs):
            # Randomize order of samples
            shuffled_indices = shuffle(
                np.array(range(0, labels.size)), random_state=model.random_state_
            )

            # Divide the shuffled indices into batches (not necessarily equal size,
            # see documentation of numpy.array_split).
            batches = np.array_split(
                shuffled_indices,
                list(range(batch_size, labels.size, batch_size)),
                axis=0,
```

(continues on next page)

(continued from previous page)

```
            )

            # Update step size using a simple annealing strategy
            step_size = self.step_size / (1 + i_run / self.max_runs)

            for i_batch in batches:
                # Select the data
                batch = data[i_batch, :]
                batch_labels = labels[i_batch]

                # Compute objective gradient
                objective_gradient = self.objective.gradient(model, batch, batch_
↪labels)

                # Multiply each param by its given step_size
                model.mul_step_size(step_size, objective_gradient)

                # Update the model by subtracting the objective-gradient (descent)␣
↪from the
                # current models variables, e.g., (prototypes, omega) in case of GMLVQ
                model.set_variables(
                    np.subtract(  # returns out=objective_gradient
                        model.get_variables(),
                        objective_gradient,
                        out=objective_gradient,
                    )
                )

            # Call the callback function if provided with updated values.
            if self.callback is not None:
                state = _update_state(
                    STATE_KEYS,
                    variables=np.copy(model.get_variables()),
                    nit=i_run + 1,
                    fun=self.objective(model, data, labels),
                    step_size=step_size,
                )
                # Simply return (stop the solver process) when callback returns true.
                if self.callback(state):
                    return
```

The CustomSteepestGradientDescent above, accompanied with some tests and documentation, would make a great addition to the sklvq package. However, it can also directly be passed to the algorithm. Some other solvers might require more functionality not supported by the models, this can be added dynamically to the model instances or by extending the required model and creating a custom model class.

```
data, labels = load_iris(return_X_y=True)

model = GLVQ(
    solver_type=CustomSteepestGradientDescent,
    distance_type="squared-euclidean",
    activation_type="sigmoid",
    activation_params={"beta": 2},
)

model.fit(data, labels)
```

(continues on next page)

(continued from previous page)

```
# Predict the labels using the trained model
predicted_labels = model.predict(data)

# Print a classification report (sklearn)
print(classification_report(labels, predicted_labels))
```

Out:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.96      0.94      0.95        50
           2       0.94      0.96      0.95        50

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

Total running time of the script: (0 minutes 0.190 seconds)

INDEX

Symbols

A

B

D

E

F

G

I

L

M

N

P

R

S

T

W