PyLMNN Documentation

Release 1.6.3

John Chiotellis

Nov 06, 2018

Contents:

1 PyLMNN **1**
- 1.1 Installation . . . 1
- 1.2 Dependencies . . . 1
- 1.3 Optional dependencies . . . 2
- 1.4 Usage . . . 2
- 1.5 References . . . 2
- 1.6 License and Contact . . . 3

2 API Reference **5**
- 2.1 pylmnn.lmnn module . . . 5
- 2.2 pylmnn.utils module . . . 9
- 2.3 pylmnn.bayesopt module . . . 9

3 Indices and tables **11**

Python Module Index **13**

CHAPTER 1

PyLMNN

PyLMNN is an implementation of the *Large Margin Nearest Neighbor* algorithm for metric learning in pure python.

This implementation follows closely the original MATLAB code by Kilian Weinberger found at https://bitbucket.org/mlcircus/lmnn. This version solves the unconstrained optimisation problem and finds a linear transformation using L-BFGS as the backend optimizer.

This package can also find optimal hyper-parameters for LMNN via Bayesian Optimization using the excellent GPyOpt package.

1.1 Installation

The code was developed in python 3.5 under Ubuntu 16.04 and was also tested under Ubuntu 18.04 and python 3.6. You can clone the repo with:

```
git clone https://github.com/johny-c/pylmnn.git
```

or install it via pip:

```
pip3 install pylmnn
```

1.2 Dependencies

- numpy>=1.11.2
- scipy>=0.18.1
- scikit_learn>=0.18.1

1.3 Optional dependencies

In case you want to use the hyperparameter optimization module, you should also install:

- GPy>=1.5.6
- GPyOpt>=1.0.3

1.4 Usage

Here is a minimal use case:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

from pylmnn import LargeMarginNearestNeighbor as LMNN


# Load a data set
X, y = load_iris(return_X_y=True)

# Split in training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.7, stratify=y,␣
↪random_state=42)

# Set up the hyperparameters
k_train, k_test, n_components, max_iter = 3, 3, X.shape[1], 180

# Instantiate the metric learner
lmnn = LMNN(n_neighbors=k_train, max_iter=max_iter, n_components=n_components)

# Train the metric learner
lmnn.fit(X_train, y_train)

# Fit the nearest neighbors classifier
knn = KNeighborsClassifier(n_neighbors=k_test)
knn.fit(lmnn.transform(X_train), y_train)

# Compute the k-nearest neighbor test accuracy after applying the learned␣
↪transformation
lmnn_acc = knn.score(lmnn.transform(X_test), y_test)
print('LMNN accuracy on test set of {} points: {:.4f}'.format(X_test.shape[0], lmnn_
↪acc))
```

You can check the examples directory for a demonstration of how to use the code with different datasets and how to estimate good hyperparameters with Bayesian Optimisation.

Documentation can also be found at http://pylmnn.readthedocs.io/en/latest/ .

1.5 References

If you use this code in your work, please cite the following publication.

```
@ARTICLE{weinberger09distance,
    title={Distance metric learning for large margin nearest neighbor classification},
    author={Weinberger, K.Q. and Saul, L.K.},
    journal={The Journal of Machine Learning Research},
    volume={10},
    pages={207--244},
    year={2009},
    publisher={MIT Press}
}
```

1.6 License and Contact

This work is released under the 3-Clause BSD License.

Contact **John Chiotellis** :envelope: for questions, comments and reporting bugs.

CHAPTER 2

API Reference

2.1 pylmnn.lmnn module

Large Margin Nearest Neighbor Classification

class `pylmnn.lmnn.`**`LargeMarginNearestNeighbor`**(*n_neighbors=3, n_components=None, init='pca', warm_start=False, max_impostors=500000, neighbors_params=None, weight_push_loss=0.5, impostor_store='auto', max_iter=50, tol=1e-05, callback=None, store_opt_result=False, verbose=0, random_state=None, n_jobs=1*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Distance metric learning for large margin classification.

n_neighbors [int, optional (default=3)] Number of neighbors to use as target neighbors for each sample.

n_components [int, optional (default=None)] Preferred dimensionality of the embedding. If None it is inferred from `init`.

init [string or numpy array, optional (default='pca')] Initialization of the linear transformation. Possible options are 'pca', 'identity' and a numpy array of shape (n_features_a, n_features_b).

pca: `n_components` many principal components of the inputs passed to *`fit()`* will be used to initialize the transformation.

identity: If `n_components` is strictly smaller than the dimensionality of the inputs passed to *`fit()`*, the identity matrix will be truncated to the first `n_components` rows.

numpy array: n_features_b must match the dimensionality of the inputs passed to *`fit()`* and n_features_a must be less than or equal to that. If `n_components` is not None, n_features_a must match it.

warm_start [bool, optional, (default=False)] If True and *fit()* has been called before, the solution of the previous call to *fit()* is used as the initial linear transformation (`n_components` and `init` will be ignored).

max_impostors [int, optional (default=500000)] Maximum number of impostors to consider per iteration. In the worst case this will allow `max_impostors * n_neighbors` constraints to be active.

neighbors_params [dict, optional (default=None)] Parameters to pass to a `neighbors.NearestNeighbors` instance - apart from `n_neighbors` - that will be used to select the target neighbors.

weight_push_loss [float, optional (default=0.5)] A float in (0, 1], weighting the push loss. This is parameter μ in the journal paper (See references below). In practice, the objective function will be normalized so that the push loss has weight 1 and hence the pull loss has weight $(1 - \mu)/\mu$.

impostor_store [str ['auto'|'list'|'sparse'], optional]

list : Three lists will be used to store the indices of reference samples, the indices of their impostors and the (squared) distances between the (sample, impostor) pairs.

sparse : A sparse indicator matrix will be used to store the (sample, impostor) pairs. The (squared) distances to the impostors will be computed twice (once to determine the impostors and once to be stored), but this option tends to be faster than 'list' as the size of the data set increases.

auto : Will attempt to decide the most appropriate choice of data structure based on the values passed to *fit()*.

max_iter [int, optional (default=50)] Maximum number of iterations in the optimization.

tol [float, optional (default=1e-5)] Convergence tolerance for the optimization.

callback [callable, optional (default=None)] If not None, this function is called after every iteration of the optimizer, taking as arguments the current solution (transformation) and the number of iterations. This might be useful in case one wants to examine or store the transformation found after each iteration.

store_opt_result [bool, optional (default=False)] If True, the `scipy.optimize.OptimizeResult` object returned by `minimize()` of *scipy.optimize* will be stored as attribute `opt_result_`.

verbose [int, optional (default=0)] If 0, no progress messages will be printed. If 1, progress messages will be printed to stdout. If > 1, progress messages will be printed and the `iprint` parameter of `_minimize_lbfgsb()` of *scipy.optimize* will be set to `verbose - 2`.

random_state [int or numpy.RandomState or None, optional (default=None)] A pseudo random number generator object or a seed for it if int.

n_jobs [int, optional (default=1)] The number of parallel jobs to run for neighbors search. If `-1`, then the number of jobs is set to the number of CPU cores. Doesn't affect *fit()* method.

components_ [array, shape (n_components, n_features)] The linear transformation learned during fitting.

n_neighbors_ [int] The provided `n_neighbors` is decreased if it is greater than or equal to min(number of elements in each class).

n_iter_ [int] Counts the number of iterations performed by the optimizer.

opt_result_ [scipy.optimize.OptimizeResult (optional)] A dictionary of information representing the optimization result. This is stored only if `store_opt_result` is True. It contains the following attributes:

x [ndarray] The solution of the optimization.

success [bool] Whether or not the optimizer exited successfully.

status [int] Termination status of the optimizer.

message [str] Description of the cause of the termination.

fun, jac [ndarray] Values of objective function and its Jacobian.

hess_inv [scipy.sparse.linalg.LinearOperator] the product of a vector with the approximate inverse of the Hessian of the objective function..

nfev [int] Number of evaluations of the objective function..

nit [int] Number of iterations performed by the optimizer.

```
>>> from pylmnn import LargeMarginNearestNeighbor
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> lmnn = LargeMarginNearestNeighbor(n_neighbors=3, random_state=42)
>>> lmnn.fit(X_train, y_train)
LargeMarginNearestNeighbor(...)
>>> # Fit and evaluate a simple nearest neighbor classifier for comparison
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> knn.fit(X_train, y_train)
KNeighborsClassifier(...)
>>> print(knn.score(X_test, y_test))
0.933333333333
>>> # Now fit on the data transformed by the learned transformation
>>> knn.fit(lmnn.transform(X_train), y_train)
KNeighborsClassifier(...)
>>> print(knn.score(lmnn.transform(X_test), y_test))
0.971428571429
```

Warning: Exact floating-point reproducibility is generally not guaranteed (unless special care is taken with library and compiler options). As a consequence, the transformations computed in 2 identical runs of LargeMarginNearestNeighbor can differ from each other. This can happen even before the optimizer is called if initialization with PCA is used (init='pca').

`fit` (*X*, *y*)

Fit the model according to the given training data.

X [array-like, shape (n_samples, n_features)] The training samples.

y [array-like, shape (n_samples,)] The corresponding training labels.

self [object] returns a trained LargeMarginNearestNeighbor model.

`transform` (*X*)

Applies the learned transformation to the given data.

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

X_embedded: array, shape (n_samples, n_components) The data samples transformed.

NotFittedError If *`fit()`* has not been called before.

`pylmnn.lmnn.make_lmnn_pipeline`(*n_neighbors=3, n_components=None, init='pca', warm_start=False, max_impostors=500000, neighbors_params=None, weight_push_loss=0.5, impostor_store='auto', max_iter=50, tol=1e-05, callback=None, store_opt_result=False, verbose=0, random_state=None, n_jobs=1, n_neighbors_predict=None, weights='uniform', algorithm='auto', leaf_size=30, n_jobs_predict=None, **kwargs*)

Constructs a LargeMarginNearestNeighbor - KNeighborsClassifier pipeline.

See LargeMarginNearestNeighbor module documentation for details.

n_neighbors_predict [int, optional (default=None)] The number of neighbors to use during prediction. If None (default) the value of `n_neighbors` used to train the model is used.

weights [str or callable, optional (default = 'uniform')] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

n_jobs_predict [int, optional (default=None)] The number of parallel jobs to run for neighbors search during prediction. If None (default), then the value of `n_jobs` is used.

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

lmnn_pipe [Pipeline] A Pipeline instance with two steps: a `LargeMarginNearestNeighbor` instance that is used to fit the model and a `KNeighborsClassifier` instance that is used for prediction.

```
>>> from pylmnn import make_lmnn_pipeline
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
```

(continues on next page)

(continued from previous page)

```
>>> lmnn_pipe = make_lmnn_pipeline(n_neighbors=3, n_neighbors_predict=3,
... random_state=42)
>>> lmnn_pipe.fit(X_train, y_train)
Pipeline(...)
>>> print(lmnn_pipe.score(X_test, y_test))
0.971428571429
```

2.2 pylmnn.utils module

2.3 pylmnn.bayesopt module

`pylmnn.bayesopt.`**`find_hyperparams`**`(`*X_train, y_train, X_valid, y_valid, params=None, max_bopt_iter=12*`)`

Find the best hyperparameters for LMNN using Bayesian Optimisation.

X_train [array_like] An array of training samples with shape (n_samples, n_features).

y_train [array_like] An array of training labels with shape (n_samples,).

X_valid [array_like] An array of validation samples with shape (m_samples, n_features).

y_valid [array_like] An array of validation labels with shape (m_samples,).

params [dict] A dictionary of parameters to be passed to the LargeMarginNearestNeighbor classifier instance.

max_bopt_iter [int] Maximum number of parameter configurations to evaluate (Default value = 12).

tuple: (int, int, int, int) The best hyperparameters found (n_neighbors, n_neighbors_predict, n_components, max_iter).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pylmnn.bayesopt, 9
pylmnn.lmnn, 5
pylmnn.utils, 9

Index

F

find_hyperparams() (in module pylmnn.bayesopt), 9
fit() (pylmnn.lmnn.LargeMarginNearestNeighbor method), 7

L

LargeMarginNearestNeighbor (class in pylmnn.lmnn), 5

M

make_lmnn_pipeline() (in module pylmnn.lmnn), 7

P

pylmnn.bayesopt (module), 9
pylmnn.lmnn (module), 5
pylmnn.utils (module), 9

T

transform() (pylmnn.lmnn.LargeMarginNearestNeighbor method), 7