d073a706b5

scikit-multiflow / src / skmultiflow / core / base.py / Jump to

smastelini Fix PEP8 problems (#247) ... History

5 contributors

679 lines (547 sloc) 23.5 KB

```
""" Base class for all estimators in scikit-multiflow """

# Some code has been copied from sklearn for compatibility.
# Following is the commit information to track the last matching version with sklearn's base.py:
# https://github.com/scikit-learn/scikit-learn/
# blob/1fe00b58949a6b9bce45e9e15eb8b9c138bd6a2e/sklearn/base.py

from abc import ABCMeta, abstractmethod

import copy
import warnings
from collections import defaultdict
import inspect
import re

import numpy as np

from .._version import __version__

_DEFAULT_TAGS = {
    'non_deterministic': False,
    'requires_positive_data': False,
    'X_types': ['2darray'],
    'poor_score': False,
    'no_validation': False,
    'multioutput': False,
    "allow_nan": False,
    'stateless': False,
    'multilabel': False,
    '_skip_test': False,
    'multioutput_only': False}


def clone(estimator, safe=True):
```

```
    """Constructs a new estimator with the same parameters.

    Clone does a deep copy of the model in an estimator
    without actually copying attached data. It yields a new estimator
    with the same parameters that has not been fit on any data.

    Parameters
    ----------
    estimator : estimator object, or list, tuple or set of objects
        The estimator or group of estimators to be cloned

    safe : boolean, optional
        If safe is false, clone will fall back to a deep copy on objects
        that are not estimators.

    Notes
    -----
    Taken from sklearn for compatibility.
    """
    estimator_type = type(estimator)
    # XXX: not handling dictionaries
    if estimator_type in (list, tuple, set, frozenset):
        return estimator_type([clone(e, safe=safe) for e in estimator])
    elif not hasattr(estimator, 'get_params') or isinstance(estimator, type):
        if not safe:
            return copy.deepcopy(estimator)
        else:
            raise TypeError("Cannot clone object '%s' (type %s): "
                            "it does not seem to be a valid estimator "
                            "as it does not implement a 'get_params' methods."
                            % (repr(estimator), type(estimator)))
    klass = estimator.__class__
    new_object_params = estimator.get_params(deep=False)
    for name, param in new_object_params.items():
        new_object_params[name] = clone(param, safe=False)
    new_object = klass(**new_object_params)
    params_set = new_object.get_params(deep=False)

    # quick sanity check of the parameters of the clone
    for name in new_object_params:
        param1 = new_object_params[name]
        param2 = params_set[name]
        if param1 is not param2:
            raise RuntimeError('Cannot clone object %s, as the constructor '
                               'either does not set or modifies parameter %s' %
                               (estimator, name))
    return new_object


def _pprint(params, offset=0, printer=repr):
    """Pretty print the dictionary 'params'

    Parameters
    ----------
```

```
    params : dict
        The dictionary to pretty print

    offset : int
        The offset in characters to add at the begin of each line.

    printer : callable
        The function to convert entries to strings, typically
        the builtin str or repr

    Notes
    -----
    Taken from sklearn for compatibility.
    """
    # Do a multi-line justified repr:
    options = np.get_printoptions()
    np.set_printoptions(precision=5, threshold=64, edgeitems=2)
    params_list = list()
    this_line_length = offset
    line_sep = ',\n' + (1 + offset // 2) * ' '
    for i, (k, v) in enumerate(sorted(params.items())):
        if type(v) is float:
            # use str for representing floating point numbers
            # this way we get consistent representation across
            # architectures and versions.
            this_repr = '%s=%s' % (k, str(v))
        else:
            # use repr of the rest
            this_repr = '%s=%s' % (k, printer(v))
        if len(this_repr) > 500:
            this_repr = this_repr[:300] + '...' + this_repr[-100:]
        if i > 0:
            if (this_line_length + len(this_repr) >= 75 or '\n' in this_repr):
                params_list.append(line_sep)
                this_line_length = len(line_sep)
            else:
                params_list.append(', ')
                this_line_length += 2
        params_list.append(this_repr)
        this_line_length += len(this_repr)

    np.set_printoptions(**options)
    lines = ''.join(params_list)
    # Strip trailing space to avoid nightmare in doctests
    lines = '\n'.join(line.rstrip(' ') for line in lines.split('\n'))
    return lines


def _update_if_consistent(dict1, dict2):
    common_keys = set(dict1.keys()).intersection(dict2.keys())
    for key in common_keys:
        if dict1[key] != dict2[key]:
            raise TypeError("Inconsistent values for tag {}: {} != {}".format(
                key, dict1[key], dict2[key]
```

```
            ))
    dict1.update(dict2)
    return dict1


class BaseEstimator:
    """Base Estimator class for compatibility with scikit-learn.

    Notes
    -----
    * All estimators should specify all the parameters that can be set
      at the class level in their ``__init__`` as explicit keyword
      arguments (no ``*args`` or ``**kwargs``).
    * Taken from sklearn for compatibility.
    """

    @classmethod
    def _get_param_names(cls):
        """Get parameter names for the estimator"""
        # fetch the constructor or the original constructor before
        # deprecation wrapping if any
        init = getattr(cls.__init__, 'deprecated_original', cls.__init__)
        if init is object.__init__:
            # No explicit constructor to introspect
            return []

        # introspect the constructor arguments to find the model parameters
        # to represent
        init_signature = inspect.signature(init)
        # Consider the constructor parameters excluding 'self'
        parameters = [p for p in init_signature.parameters.values()
                      if p.name != 'self' and p.kind != p.VAR_KEYWORD]
        for p in parameters:
            if p.kind == p.VAR_POSITIONAL:
                raise RuntimeError("scikit-multiflow estimators should always "
                                   "specify their parameters in the signature"
                                   " of their __init__ (no varargs)."
                                   " %s with constructor %s doesn't "
                                   " follow this convention."
                                   % (cls, init_signature))
        # Extract and sort argument names excluding 'self'
        return sorted([p.name for p in parameters])

    def get_params(self, deep=True):
        """Get parameters for this estimator.

        Parameters
        ----------
        deep : boolean, optional
            If True, will return the parameters for this estimator and
            contained subobjects that are estimators.

        Returns
        -------
```

```
        params : mapping of string to any
            Parameter names mapped to their values.
        """
        out = dict()
        for key in self._get_param_names():
            value = getattr(self, key, None)
            if deep and hasattr(value, 'get_params'):
                deep_items = value.get_params().items()
                out.update((key + '__' + k, val) for k, val in deep_items)
            out[key] = value
        return out

    def set_params(self, **params):
        """Set the parameters of this estimator.

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

        Returns
        -------
        self
        """
        if not params:
            # Simple optimization to gain speed (inspect is slow)
            return self
        valid_params = self.get_params(deep=True)

        nested_params = defaultdict(dict)  # grouped by prefix
        for key, value in params.items():
            key, delim, sub_key = key.partition('__')
            if key not in valid_params:
                raise ValueError('Invalid parameter %s for estimator %s. '
                                 'Check the list of available parameters '
                                 'with `estimator.get_params().keys()`.' %
                                 (key, self))

            if delim:
                nested_params[key][sub_key] = value
            else:
                setattr(self, key, value)
                valid_params[key] = value

        for key, sub_params in nested_params.items():
            valid_params[key].set_params(**sub_params)

        return self

    def __repr__(self, N_CHAR_MAX=700):
        # N_CHAR_MAX is the (approximate) maximum number of non-blank
        # characters to render. We pass it as an optional parameter to ease
        # the tests.

```

```
        from ..utils._pprint import _EstimatorPrettyPrinter

        N_MAX_ELEMENTS_TO_SHOW = 30  # number of elements to show in sequences

        # use ellipsis for sequences with a lot of elements
        pp = _EstimatorPrettyPrinter(
            compact=True, indent=1, indent_at_name=True,
            n_max_elements_to_show=N_MAX_ELEMENTS_TO_SHOW)

        repr_ = pp.pformat(self)

        # Use bruteforce ellipsis when there are a lot of non-blank characters
        n_nonblank = len(''.join(repr_.split()))
        if n_nonblank > N_CHAR_MAX:
            lim = N_CHAR_MAX // 2  # apprx number of chars to keep on both ends
            regex = r'^(\s*\S){%d}' % lim
            # The regex '^(\s*\S){%d}' % n
            # matches from the start of the string until the nth non-blank
            # character:
            # - ^ matches the start of string
            # - (pattern){n} matches n repetitions of pattern
            # - \s*\S matches a non-blank char following zero or more blanks
            left_lim = re.match(regex, repr_).end()
            right_lim = re.match(regex, repr_[::-1]).end()

            if '\n' in repr_[left_lim:-right_lim]:
                # The left side and right side aren't on the same line.
                # To avoid weird cuts, e.g.:
                # categoric...ore',
                # we need to start the right side with an appropriate newline
                # character so that it renders properly as:
                # categoric...
                # handle_unknown='ignore',
                # so we add [^\n]*\n which matches until the next \n
                regex += r'[^\n]*\n'
                right_lim = re.match(regex, repr_[::-1]).end()

            ellipsis = '...'
            if left_lim + len(ellipsis) < len(repr_) - right_lim:
                # Only add ellipsis if it results in a shorter repr
                repr_ = repr_[:left_lim] + '...' + repr_[-right_lim:]

        return repr_

    def __getstate__(self):
        try:
            state = super().__getstate__()
        except AttributeError:
            state = self.__dict__.copy()

        if type(self).__module__.startswith('skmultiflow.'):
            return dict(state.items(), _skmultiflow_version=__version__)
        else:
            return state
```

```
305
306     def __setstate__(self, state):
307         if type(self).__module__.startswith('skmultiflow.'):
308             pickle_version = state.pop("_skmultiflow_version", "pre-0.18")
309             if pickle_version != __version__:
310                 warnings.warn(
311                     "Trying to unpickle estimator {0} from version {1} when "
312                     "using version {2}. This might lead to breaking code or "
313                     "invalid results. Use at your own risk.".format(
314                         self.__class__.__name__, pickle_version, __version__),
315                     UserWarning)
316         try:
317             super().__setstate__(state)
318         except AttributeError:
319             self.__dict__.update(state)
320
321     def _get_tags(self):
322         collected_tags = {}
323         for base_class in inspect.getmro(self.__class__):
324             if (hasattr(base_class, '_more_tags') and base_class != self.__class__):
325                 more_tags = base_class._more_tags(self)
326                 collected_tags = _update_if_consistent(collected_tags,
327                                                        more_tags)
328         if hasattr(self, '_more_tags'):
329             more_tags = self._more_tags()
330             collected_tags = _update_if_consistent(collected_tags, more_tags)
331         tags = _DEFAULT_TAGS.copy()
332         tags.update(collected_tags)
333         return tags
334
335
336 class BaseSKMObject(BaseEstimator):
337     """Base class for most objects in scikit-multiflow
338
339         Notes
340         -----
341         This class provides additional functionality not available in the base estimator
342         from scikit-learn
343     """
344     def reset(self):
345         """ Resets the estimator to its initial state.
346
347         Returns
348         -------
349         self
350
351         """
352         # non-optimized default implementation; override if a better
353         # method is possible for a given object
354         command = ''.join([line.strip() for line in self.__repr__().split()])
355         command = command.replace(str(self.__class__.__name__), 'self.__init__')
356         exec(command)
357
358     def get_info(self):
```

```
        """ Collects and returns the information about the configuration of the estimator

        Returns
        -------
        string
            Configuration of the estimator.
        """
        return self.__repr__()


class ClassifierMixin(metaclass=ABCMeta):
    """Mixin class for all classifiers in scikit-multiflow."""
    _estimator_type = "classifier"

    def fit(self, X, y, classes=None, sample_weight=None):
        """ Fit the model.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The features to train the model.

        y: numpy.ndarray of shape (n_samples, n_targets)
            An array-like with the class labels of all samples in X.

        classes: numpy.ndarray, optional (default=None)
            Contains all possible/known class labels. Usage varies depending
            on the learning method.

        sample_weight: numpy.ndarray, optional (default=None)
            Samples weight. If not provided, uniform weights are assumed.
            Usage varies depending on the learning method.

        Returns
        -------
        self

        """
        # non-optimized default implementation; override if a better
        # method is possible for a given classifier
        self.partial_fit(X, y, classes=classes, sample_weight=sample_weight)

        return self

    @abstractmethod
    def partial_fit(self, X, y, classes=None, sample_weight=None):
        """ Partially (incrementally) fit the model.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The features to train the model.

        y: numpy.ndarray of shape (n_samples)
```

```
            An array-like with the class labels of all samples in X.

        classes: numpy.ndarray, optional (default=None)
            Array with all possible/known class labels. Usage varies depending
            on the learning method.

        sample_weight: numpy.ndarray of shape (n_samples), optional (default=None)
            Samples weight. If not provided, uniform weights are assumed.
            Usage varies depending on the learning method.

        Returns
        -------
        self

        """
        raise NotImplementedError

    @abstractmethod
    def predict(self, X):
        """ Predict classes for the passed data.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The set of data samples to predict the class labels for.

        Returns
        -------
        A numpy.ndarray with all the predictions for the samples in X.

        """
        raise NotImplementedError

    @abstractmethod
    def predict_proba(self, X):
        """ Estimates the probability of each sample in X belonging to each of the class-labels.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The matrix of samples one wants to predict the class probabilities for.

        Returns
        -------
        A numpy.ndarray of shape (n_samples, n_labels), in which each outer entry is associated
        with the X entry of the same index. And where the list in index [i] contains
        len(self.target_values) elements, each of which represents the probability that
        the i-th sample of X belongs to a certain class-label.

        """
        raise NotImplementedError

    def score(self, X, y, sample_weight=None):
        """Returns 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, shape = (n_samples, n_features)
            Test samples.

        y : array-like, shape = (n_samples) or (n_samples, n_outputs)
            True labels for X.

        sample_weight : array-like, shape = [n_samples], optional
            Sample weights.

        Returns
        -------
        score : float
            Mean accuracy of self.predict(X) wrt. y.

        """
        from sklearn.metrics import accuracy_score
        return accuracy_score(y, self.predict(X), sample_weight=sample_weight)


class RegressorMixin(metaclass=ABCMeta):
    """Mixin class for all regression estimators in scikit-multiflow."""
    _estimator_type = "regressor"

    def fit(self, X, y, sample_weight=None):
        """ Fit the model.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The features to train the model.

        y: numpy.ndarray of shape (n_samples, n_targets)
            An array-like with the target values of all samples in X.

        sample_weight: numpy.ndarray, optional (default=None)
            Samples weight. If not provided, uniform weights are assumed. Usage varies
            depending on the learning method.

        Returns
        -------
        self

        """
        # non-optimized default implementation; override if a better
        # method is possible for a given regressor
        self.partial_fit(X, y, sample_weight=sample_weight)

```

```
        return self

    @abstractmethod
    def partial_fit(self, X, y, sample_weight=None):
        """ Partially (incrementally) fit the model.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The features to train the model.

        y: numpy.ndarray of shape (n_samples)
            An array-like with the target values of all samples in X.

        sample_weight: numpy.ndarray of shape (n_samples), optional (default=None)
            Samples weight. If not provided, uniform weights are assumed. Usage varies
            depending on the learning method.

        Returns
        -------
        self

        """
        raise NotImplementedError

    @abstractmethod
    def predict(self, X):
        """ Predict target values for the passed data.

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The set of data samples to predict the target values for.

        Returns
        -------
        A numpy.ndarray with all the predictions for the samples in X.

        """
        raise NotImplementedError

    @abstractmethod
    def predict_proba(self, X):
        """ Estimates the probability for probabilistic/bayesian regressors

        Parameters
        ----------
        X : numpy.ndarray of shape (n_samples, n_features)
            The matrix of samples one wants to predict the probabilities for.

        Returns
        -------
        numpy.ndarray

```

```
        """
        raise NotImplementedError

    def score(self, X, y, sample_weight=None):
        """Returns the coefficient of determination R^2 of the prediction.

        The coefficient R^2 is defined as (1 - u/v), where u is the residual
        sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
        sum of squares ((y_true - y_true.mean()) ** 2).sum().
        The best possible score is 1.0 and it can be negative (because the
        model can be arbitrarily worse). A constant model that always
        predicts the expected value of y, disregarding the input features,
        would get a R^2 score of 0.0.

        Parameters
        ----------
        X : array-like, shape = (n_samples, n_features)
            Test samples. For some estimators this may be a
            precomputed kernel matrix instead, shape = (n_samples,
            n_samples_fitted], where n_samples_fitted is the number of
            samples used in the fitting for the estimator.

        y : array-like, shape = (n_samples) or (n_samples, n_outputs)
            True values for X.

        sample_weight : array-like, shape = [n_samples], optional
            Sample weights.

        Returns
        -------
        score : float
            R^2 of self.predict(X) wrt. y.

        Notes
        -----
        The R2 score used when calling ``score`` on a regressor will use
        ``multioutput='uniform_average'`` from version 0.23 to keep consistent
        with `metrics.r2_score`. This will influence the ``score`` method of
        all the multioutput regressors (except for
        `multioutput.MultiOutputRegressor`). To specify the default value
        manually and avoid the warning, please either call `metrics.r2_score`
        directly or make a custom scorer with `metrics.make_scorer` (the
        built-in scorer ``'r2'`` uses ``multioutput='uniform_average'``).
        """

        from sklearn.metrics import r2_score
        from sklearn.metrics.regression import _check_reg_targets
        y_pred = self.predict(X)
        # XXX: Remove the check in 0.23
        y_type, _, _, _ = _check_reg_targets(y, y_pred, None)
        if y_type == 'continuous-multioutput':
            warnings.warn("The default value of multioutput (not exposed in "
                          "score method) will change from 'variance_weighted' "
                          "to 'uniform_average' in 0.23 to keep consistent "
```

```
                          "with 'metrics.r2_score'. To specify the default "
                          "value manually and avoid the warning, please "
                          "either call 'metrics.r2_score' directly or make a "
                          "custom scorer with 'metrics.make_scorer' (the "
                          "built-in scorer 'r2' uses "
                          "multioutput='uniform_average').", FutureWarning)
        return r2_score(y, y_pred, sample_weight=sample_weight,
                        multioutput='variance_weighted')


class MetaEstimatorMixin(object):
    """Mixin class for all meta estimators in scikit-multiflow."""
    _required_parameters = ["estimator"]


class MultiOutputMixin(object):
    """Mixin to mark estimators that support multioutput."""
    def _more_tags(self):
        return {'multioutput': True}


def is_classifier(estimator):
    """Returns True if the given estimator is (probably) a classifier.

    Parameters
    ----------
    estimator : object
        Estimator object to test.

    Returns
    -------
    out : bool
        True if estimator is a classifier and False otherwise.
    """
    return getattr(estimator, "_estimator_type", None) == "classifier"


def is_regressor(estimator):
    """Returns True if the given estimator is (probably) a regressor.

    Parameters
    ----------
    estimator : object
        Estimator object to test.

    Returns
    -------
    out : bool
        True if estimator is a regressor and False otherwise.
    """
    return getattr(estimator, "_estimator_type", None) == "regressor"
```