Source code for pyspark.ml.evaluation

```
#
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from abc import abstractmethod, ABCMeta

from pyspark import since
from pyspark.ml.wrapper import JavaWrapper
from pyspark.ml.param import Param, Params
from pyspark.ml.param.shared import HasLabelCol, HasPredictionCol, HasRawPredictionCol
from pyspark.ml.util import keyword_only
from pyspark.mllib.common import inherit_doc

__all__ = ['Evaluator', 'BinaryClassificationEvaluator', 'RegressionEvaluator',
           'MulticlassClassificationEvaluator']


@inherit_doc
class Evaluator(Params):                                                    [docs]
    """
    Base class for evaluators that compute metrics from predictions.

    .. versionadded:: 1.4.0
    """

    __metaclass__ = ABCMeta

    @abstractmethod
```

```
    def _evaluate(self, dataset):
        """
        Evaluates the output.

        :param dataset: a dataset that contains labels/observations and
               predictions
        :return: metric
        """
        raise NotImplementedError()

    @since("1.4.0")
    def evaluate(self, dataset, params=None):                                    [docs]
        """
        Evaluates the output with optional parameters.

        :param dataset: a dataset that contains labels/observations and
                        predictions
        :param params: an optional param map that overrides embedded
                       params
        :return: metric
        """
        if params is None:
            params = dict()
        if isinstance(params, dict):
            if params:
                return self.copy(params)._evaluate(dataset)
            else:
                return self._evaluate(dataset)
        else:
            raise ValueError("Params must be a param map but got %s." % type(params))

    @since("1.5.0")
    def isLargerBetter(self):                                                    [docs]
        """
        Indicates whether the metric returned by :py:meth:`evaluate` should be maximized
        (True, default) or minimized (False).
        A given evaluator may support multiple metrics which may be maximized or minimized.
        """
        return True


@inherit_doc
class JavaEvaluator(Evaluator, JavaWrapper):
    """
    Base class for :py:class:`Evaluator`s that wrap Java/Scala
    implementations.
    """
```

```
    __metaclass__ = ABCMeta

    def _evaluate(self, dataset):
        """
        Evaluates the output.
        :param dataset: a dataset that contains labels/observations and predictions.
        :return: evaluation metric
        """
        self._transfer_params_to_java()
        return self._java_obj.evaluate(dataset._jdf)

    def isLargerBetter(self):
        self._transfer_params_to_java()
        return self._java_obj.isLargerBetter()


@inherit_doc
class BinaryClassificationEvaluator(JavaEvaluator, HasLabelCol, HasRawPredictionCol):                    [docs]
    """
    Evaluator for binary classification, which expects two input
    columns: rawPrediction and label.

    >>> from pyspark.mllib.linalg import Vectors
    >>> scoreAndLabels = map(lambda x: (Vectors.dense([1.0 - x[0], x[0]]), x[1]),
    ...    [(0.1, 0.0), (0.1, 1.0), (0.4, 0.0), (0.6, 0.0), (0.6, 1.0), (0.6, 1.0), (0.8, 1.0)])
    >>> dataset = sqlContext.createDataFrame(scoreAndLabels, ["raw", "label"])
    ...
    >>> evaluator = BinaryClassificationEvaluator(rawPredictionCol="raw")
    >>> evaluator.evaluate(dataset)
    0.70...
    >>> evaluator.evaluate(dataset, {evaluator.metricName: "areaUnderPR"})
    0.83...

    .. versionadded:: 1.4.0
    """

    # a placeholder to make it appear in the generated doc
    metricName = Param(Params._dummy(), "metricName",
                       "metric name in evaluation (areaUnderROC|areaUnderPR)")

    @keyword_only
    def __init__(self, rawPredictionCol="rawPrediction", labelCol="label",
                 metricName="areaUnderROC"):
        """
        __init__(self, rawPredictionCol="rawPrediction", labelCol="label", \
                 metricName="areaUnderROC")
```

```
        """
        super(BinaryClassificationEvaluator, self).__init__()
        self._java_obj = self._new_java_obj(
            "org.apache.spark.ml.evaluation.BinaryClassificationEvaluator", self.uid)
        #: param for metric name in evaluation (areaUnderROC|areaUnderPR)
        self.metricName = Param(self, "metricName",
                                "metric name in evaluation (areaUnderROC|areaUnderPR)")
        self._setDefault(rawPredictionCol="rawPrediction", labelCol="label",
                         metricName="areaUnderROC")
        kwargs = self.__init__._input_kwargs
        self._set(**kwargs)

    @since("1.4.0")
    def setMetricName(self, value):                                                    [docs]
        """
        Sets the value of :py:attr:`metricName`.
        """
        self._paramMap[self.metricName] = value
        return self

    @since("1.4.0")
    def getMetricName(self):                                                           [docs]
        """
        Gets the value of metricName or its default value.
        """
        return self.getOrDefault(self.metricName)

    @keyword_only
    @since("1.4.0")
    def setParams(self, rawPredictionCol="rawPrediction", labelCol="label",            [docs]
                  metricName="areaUnderROC"):
        """
        setParams(self, rawPredictionCol="rawPrediction", labelCol="label", \
                  metricName="areaUnderROC")
        Sets params for binary classification evaluator.
        """
        kwargs = self.setParams._input_kwargs
        return self._set(**kwargs)


@inherit_doc
class RegressionEvaluator(JavaEvaluator, HasLabelCol, HasPredictionCol):           [docs]
    """
    Evaluator for Regression, which expects two input
    columns: prediction and label.

    >>> scoreAndLabels = [(-28.98343821, -27.0), (20.21491975, 21.5),
```

```
...   (-25.98418959, -22.0), (30.69731842, 33.0), (74.69283752, 71.0)]
>>> dataset = sqlContext.createDataFrame(scoreAndLabels, ["raw", "label"])
...
>>> evaluator = RegressionEvaluator(predictionCol="raw")
>>> evaluator.evaluate(dataset)
2.842...
>>> evaluator.evaluate(dataset, {evaluator.metricName: "r2"})
0.993...
>>> evaluator.evaluate(dataset, {evaluator.metricName: "mae"})
2.649...

.. versionadded:: 1.4.0
"""
# Because we will maximize evaluation value (ref: `CrossValidator`),
# when we evaluate a metric that is needed to minimize (e.g., `"rmse"`, `"mse"`, `"mae"`),
# we take and output the negative of this metric.
metricName = Param(Params._dummy(), "metricName",
                   "metric name in evaluation (mse|rmse|r2|mae)")

@keyword_only
def __init__(self, predictionCol="prediction", labelCol="label",
             metricName="rmse"):
    """
    __init__(self, predictionCol="prediction", labelCol="label", \
             metricName="rmse")
    """
    super(RegressionEvaluator, self).__init__()
    self._java_obj = self._new_java_obj(
        "org.apache.spark.ml.evaluation.RegressionEvaluator", self.uid)
    #: param for metric name in evaluation (mse|rmse|r2|mae)
    self.metricName = Param(self, "metricName",
                            "metric name in evaluation (mse|rmse|r2|mae)")
    self._setDefault(predictionCol="prediction", labelCol="label",
                     metricName="rmse")
    kwargs = self.__init__._input_kwargs
    self._set(**kwargs)

@since("1.4.0")
def setMetricName(self, value):                                    [docs]
    """
    Sets the value of :py:attr:`metricName`.
    """
    self._paramMap[self.metricName] = value
    return self

@since("1.4.0")
def getMetricName(self):                                           [docs]
```

```
        """
        Gets the value of metricName or its default value.
        """
        return self.getOrDefault(self.metricName)

    @keyword_only
    @since("1.4.0")
    def setParams(self, predictionCol="prediction", labelCol="label",                    [docs]
                  metricName="rmse"):
        """
        setParams(self, predictionCol="prediction", labelCol="label", \
                  metricName="rmse")
        Sets params for regression evaluator.
        """
        kwargs = self.setParams._input_kwargs
        return self._set(**kwargs)


@inherit_doc
class MulticlassClassificationEvaluator(JavaEvaluator, HasLabelCol, HasPredictionCol):    [docs]
    """
    Evaluator for Multiclass Classification, which expects two input
    columns: prediction and label.
    >>> scoreAndLabels = [(0.0, 0.0), (0.0, 1.0), (0.0, 0.0),
    ...     (1.0, 0.0), (1.0, 1.0), (1.0, 1.0), (1.0, 1.0), (2.0, 2.0), (2.0, 0.0)]
    >>> dataset = sqlContext.createDataFrame(scoreAndLabels, ["prediction", "label"])
    ...
    >>> evaluator = MulticlassClassificationEvaluator(predictionCol="prediction")
    >>> evaluator.evaluate(dataset)
    0.66...
    >>> evaluator.evaluate(dataset, {evaluator.metricName: "precision"})
    0.66...
    >>> evaluator.evaluate(dataset, {evaluator.metricName: "recall"})
    0.66...

    .. versionadded:: 1.5.0
    """
    # a placeholder to make it appear in the generated doc
    metricName = Param(Params._dummy(), "metricName",
                       "metric name in evaluation "
                       "(f1|precision|recall|weightedPrecision|weightedRecall)")

    @keyword_only
    def __init__(self, predictionCol="prediction", labelCol="label",
                 metricName="f1"):
        """
        __init__(self, predictionCol="prediction", labelCol="label", \
```

```
                  metricName="f1")
        """
        super(MulticlassClassificationEvaluator, self).__init__()
        self._java_obj = self._new_java_obj(
            "org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator", self.uid)
        # param for metric name in evaluation (f1|precision|recall|weightedPrecision|weightedRecall)
        self.metricName = Param(self, "metricName",
                                "metric name in evaluation"
                                " (f1|precision|recall|weightedPrecision|weightedRecall)")
        self._setDefault(predictionCol="prediction", labelCol="label",
                         metricName="f1")
        kwargs = self.__init__._input_kwargs
        self._set(**kwargs)

    @since("1.5.0")
    def setMetricName(self, value):                                                    [docs]
        """
        Sets the value of :py:attr:`metricName`.
        """
        self._paramMap[self.metricName] = value
        return self

    @since("1.5.0")
    def getMetricName(self):                                                           [docs]
        """
        Gets the value of metricName or its default value.
        """
        return self.getOrDefault(self.metricName)

    @keyword_only
    @since("1.5.0")
    def setParams(self, predictionCol="prediction", labelCol="label",                  [docs]
                  metricName="f1"):
        """
        setParams(self, predictionCol="prediction", labelCol="label", \
                  metricName="f1")
        Sets params for multiclass classification evaluator.
        """
        kwargs = self.setParams._input_kwargs
        return self._set(**kwargs)

if __name__ == "__main__":
    import doctest
    from pyspark.context import SparkContext
    from pyspark.sql import SQLContext
    globs = globals().copy()
    # The small batch size here ensures that we see multiple batches,
```

```
    # even in these small test examples:
    sc = SparkContext("local[2]", "ml.evaluation tests")
    sqlContext = SQLContext(sc)
    globs['sc'] = sc
    globs['sqlContext'] = sqlContext
    (failure_count, test_count) = doctest.testmod(
        globs=globs, optionflags=doctest.ELLIPSIS)
    sc.stop()
    if failure_count:
        exit(-1)
```