agamemnonc DOC Balanced accuracy score adjusted doc fix (#19309)

46 contributors +29

2510 lines (2063 sloc) | 94.9 KB

```python
"""Metrics to assess performance on classification task given class prediction.

Functions named as ``*_score`` return a scalar value to maximize: the higher
the better.

Function named as ``*_error`` or ``*_loss`` return a scalar value to minimize:
the lower the better.
"""

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Arnaud Joly <a.joly@ulg.ac.be>
#          Jochen Wersdorfer <jochen@wersdoerfer.de>
#          Lars Buitinck
#          Joel Nothman <joel.nothman@gmail.com>
#          Noel Dawe <noel@dawe.me>
#          Jatin Shah <jatindshah@gmail.com>
#          Saurabh Jha <saurabh.jhaa@gmail.com>
#          Bernardo Stein <bernardovstein@gmail.com>
#          Shangwu Yao <shangwuyao@gmail.com>
# License: BSD 3 clause


import warnings
import numpy as np

from scipy.sparse import coo_matrix
from scipy.sparse import csr_matrix

from ..preprocessing import LabelBinarizer
from ..preprocessing import LabelEncoder
```

```python
from ..utils import assert_all_finite
from ..utils import check_array
from ..utils import check_consistent_length
from ..utils import column_or_1d
from ..utils.multiclass import unique_labels
from ..utils.multiclass import type_of_target
from ..utils.validation import _num_samples
from ..utils.validation import _deprecate_positional_args
from ..utils.sparsefuncs import count_nonzero
from ..exceptions import UndefinedMetricWarning

from ._base import _check_pos_label_consistency


def _check_zero_division(zero_division):
    if isinstance(zero_division, str) and zero_division == "warn":
        return
    elif isinstance(zero_division, (int, float)) and zero_division in [0, 1]:
        return
    raise ValueError('Got zero_division={0}.'
                     ' Must be one of ["warn", 0, 1]'.format(zero_division))


def _check_targets(y_true, y_pred):
    """Check that y_true and y_pred belong to the same classification task.

    This converts multiclass or binary types to a common shape, and raises a
    ValueError for a mix of multilabel and multiclass targets, a mix of
    multilabel formats, for the presence of continuous-valued or multioutput
    targets, or for targets of different lengths.

    Column vectors are squeezed to 1d, while multilabel formats are returned
    as CSR sparse label indicators.

    Parameters
    ----------
    y_true : array-like

    y_pred : array-like

    Returns
    -------
    type_true : one of {'multilabel-indicator', 'multiclass', 'binary'}
        The type of the true target data, as output by
        ``utils.multiclass.type_of_target``.

    y_true : array or indicator matrix

    y_pred : array or indicator matrix
    """
    check_consistent_length(y_true, y_pred)
    type_true = type_of_target(y_true)
```

```python
    type_pred = type_of_target(y_pred)

    y_type = {type_true, type_pred}
    if y_type == {"binary", "multiclass"}:
        y_type = {"multiclass"}

    if len(y_type) > 1:
        raise ValueError("Classification metrics can't handle a mix of {0} "
                         "and {1} targets".format(type_true, type_pred))

    # We can't have more than one value on y_type => The set is no more needed
    y_type = y_type.pop()

    # No metrics support "multiclass-multioutput" format
    if (y_type not in ["binary", "multiclass", "multilabel-indicator"]):
        raise ValueError("{0} is not supported".format(y_type))

    if y_type in ["binary", "multiclass"]:
        y_true = column_or_1d(y_true)
        y_pred = column_or_1d(y_pred)
        if y_type == "binary":
            try:
                unique_values = np.union1d(y_true, y_pred)
            except TypeError as e:
                # We expect y_true and y_pred to be of the same data type.
                # If `y_true` was provided to the classifier as strings,
                # `y_pred` given by the classifier will also be encoded with
                # strings. So we raise a meaningful error
                raise TypeError(
                    f"Labels in y_true and y_pred should be of the same type. "
                    f"Got y_true={np.unique(y_true)} and "
                    f"y_pred={np.unique(y_pred)}. Make sure that the "
                    f"predictions provided by the classifier coincides with "
                    f"the true labels."
                ) from e
            if len(unique_values) > 2:
                y_type = "multiclass"

    if y_type.startswith('multilabel'):
        y_true = csr_matrix(y_true)
        y_pred = csr_matrix(y_pred)
        y_type = 'multilabel-indicator'

    return y_type, y_true, y_pred


def _weighted_sum(sample_score, sample_weight, normalize=False):
    if normalize:
        return np.average(sample_score, weights=sample_weight)
    elif sample_weight is not None:
        return np.dot(sample_score, sample_weight)
    else:
```

```python
        return sample_score.sum()


@_deprecate_positional_args
def accuracy_score(y_true, y_pred, *, normalize=True, sample_weight=None):
    """Accuracy classification score.

    In multilabel classification, this function computes subset accuracy:
    the set of labels predicted for a sample must *exactly* match the
    corresponding set of labels in y_true.

    Read more in the :ref:`User Guide <accuracy_score>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

    normalize : bool, default=True
        If ``False``, return the number of correctly classified samples.
        Otherwise, return the fraction of correctly classified samples.

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

    Returns
    -------
    score : float
        If ``normalize == True``, return the fraction of correctly
        classified samples (float), else returns the number of correctly
        classified samples (int).

        The best performance is 1 with ``normalize == True`` and the number
        of samples with ``normalize == False``.

    See Also
    --------
    jaccard_score, hamming_loss, zero_one_loss

    Notes
    -----
    In binary and multiclass classification, this function is equal
    to the ``jaccard_score`` function.

    Examples
    --------
    >>> from sklearn.metrics import accuracy_score
    >>> y_pred = [0, 2, 1, 3]
    >>> y_true = [0, 1, 2, 3]
```

```
    >>> accuracy_score(y_true, y_pred)
    0.5
    >>> accuracy_score(y_true, y_pred, normalize=False)
    2

    In the multilabel case with binary label indicators:

    >>> import numpy as np
    >>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
    0.5
    """

    # Compute accuracy for each possible representation
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    check_consistent_length(y_true, y_pred, sample_weight)
    if y_type.startswith('multilabel'):
        differing_labels = count_nonzero(y_true - y_pred, axis=1)
        score = differing_labels == 0
    else:
        score = y_true == y_pred

    return _weighted_sum(score, sample_weight, normalize)


@_deprecate_positional_args
def confusion_matrix(y_true, y_pred, *, labels=None, sample_weight=None,
                     normalize=None):
    """Compute confusion matrix to evaluate the accuracy of a classification.

    By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}`
    is equal to the number of observations known to be in group :math:`i` and
    predicted to be in group :math:`j`.

    Thus in binary classification, the count of true negatives is
    :math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is
    :math:`C_{1,1}` and false positives is :math:`C_{0,1}`.

    Read more in the :ref:`User Guide <confusion_matrix>`.

    Parameters
    ----------
    y_true : array-like of shape (n_samples,)
        Ground truth (correct) target values.

    y_pred : array-like of shape (n_samples,)
        Estimated targets as returned by a classifier.

    labels : array-like of shape (n_classes), default=None
        List of labels to index the matrix. This may be used to reorder
        or select a subset of labels.
        If ``None`` is given, those that appear at least once
        in ``y_true`` or ``y_pred`` are used in sorted order.
```

```

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

        .. versionadded:: 0.18

    normalize : {'true', 'pred', 'all'}, default=None
        Normalizes confusion matrix over the true (rows), predicted (columns)
        conditions or all the population. If None, confusion matrix will not be
        normalized.

    Returns
    -------
    C : ndarray of shape (n_classes, n_classes)
        Confusion matrix whose i-th row and j-th
        column entry indicates the number of
        samples with true label being i-th class
        and predicted label being j-th class.

    See Also
    --------
    ConfusionMatrixDisplay.from_estimator : Plot the confusion matrix
        given an estimator, the data, and the label.
    ConfusionMatrixDisplay.from_predictions : Plot the confusion matrix
        given the true and predicted labels.
    ConfusionMatrixDisplay : Confusion Matrix visualization.

    References
    ----------
    .. [1] `Wikipedia entry for the Confusion matrix
           <https://en.wikipedia.org/wiki/Confusion_matrix>`_
           (Wikipedia and other references may use a different
           convention for axes).

    Examples
    --------
    >>> from sklearn.metrics import confusion_matrix
    >>> y_true = [2, 0, 2, 2, 0, 1]
    >>> y_pred = [0, 0, 2, 2, 0, 2]
    >>> confusion_matrix(y_true, y_pred)
    array([[2, 0, 0],
           [0, 0, 1],
           [1, 0, 2]])

    >>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
    >>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
    >>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
    array([[2, 0, 0],
           [0, 0, 1],
           [1, 0, 2]])

    In the binary case, we can extract true positives, etc as follows:
```

```python

    >>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
    >>> (tn, fp, fn, tp)
    (0, 2, 1, 1)

    """
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    if y_type not in ("binary", "multiclass"):
        raise ValueError("%s is not supported" % y_type)

    if labels is None:
        labels = unique_labels(y_true, y_pred)
    else:
        labels = np.asarray(labels)
        n_labels = labels.size
        if n_labels == 0:
            raise ValueError("'labels' should contains at least one label.")
        elif y_true.size == 0:
            return np.zeros((n_labels, n_labels), dtype=int)
        elif np.all([l not in y_true for l in labels]):
            raise ValueError("At least one label specified must be in y_true")

    if sample_weight is None:
        sample_weight = np.ones(y_true.shape[0], dtype=np.int64)
    else:
        sample_weight = np.asarray(sample_weight)

    check_consistent_length(y_true, y_pred, sample_weight)

    if normalize not in ['true', 'pred', 'all', None]:
        raise ValueError("normalize must be one of {'true', 'pred', "
                         "'all', None}")

    n_labels = labels.size
    label_to_ind = {y: x for x, y in enumerate(labels)}
    # convert yt, yp into index
    y_pred = np.array([label_to_ind.get(x, n_labels + 1) for x in y_pred])
    y_true = np.array([label_to_ind.get(x, n_labels + 1) for x in y_true])

    # intersect y_pred, y_true with labels, eliminate items not in labels
    ind = np.logical_and(y_pred < n_labels, y_true < n_labels)
    y_pred = y_pred[ind]
    y_true = y_true[ind]
    # also eliminate weights of eliminated items
    sample_weight = sample_weight[ind]

    # Choose the accumulator dtype to always have high precision
    if sample_weight.dtype.kind in {'i', 'u', 'b'}:
        dtype = np.int64
    else:
        dtype = np.float64

```

```python
    cm = coo_matrix((sample_weight, (y_true, y_pred)),
                    shape=(n_labels, n_labels), dtype=dtype,
                    ).toarray()

    with np.errstate(all='ignore'):
        if normalize == 'true':
            cm = cm / cm.sum(axis=1, keepdims=True)
        elif normalize == 'pred':
            cm = cm / cm.sum(axis=0, keepdims=True)
        elif normalize == 'all':
            cm = cm / cm.sum()
        cm = np.nan_to_num(cm)

    return cm


@_deprecate_positional_args
def multilabel_confusion_matrix(y_true, y_pred, *, sample_weight=None,
                                labels=None, samplewise=False):
    """Compute a confusion matrix for each class or sample.

    .. versionadded:: 0.21

    Compute class-wise (default) or sample-wise (samplewise=True) multilabel
    confusion matrix to evaluate the accuracy of a classification, and output
    confusion matrices for each class or sample.

    In multilabel confusion matrix :math:`MCM`, the count of true negatives
    is :math:`MCM_{:,0,0}`, false negatives is :math:`MCM_{:,1,0}`,
    true positives is :math:`MCM_{:,1,1}` and false positives is
    :math:`MCM_{:,0,1}`.

    Multiclass data will be treated as if binarized under a one-vs-rest
    transformation. Returned confusion matrices will be in the order of
    sorted unique labels in the union of (y_true, y_pred).

    Read more in the :ref:`User Guide <multilabel_confusion_matrix>`.

    Parameters
    ----------
    y_true : {array-like, sparse matrix} of shape (n_samples, n_outputs) or \
            (n_samples,)
        Ground truth (correct) target values.

    y_pred : {array-like, sparse matrix} of shape (n_samples, n_outputs) or \
            (n_samples,)
        Estimated targets as returned by a classifier.

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

    labels : array-like of shape (n_classes,), default=None
```

```
        A list of classes or column indices to select some (or to force
        inclusion of classes absent from the data).

    samplewise : bool, default=False
        In the multilabel case, this calculates a confusion matrix per sample.

    Returns
    -------
    multi_confusion : ndarray of shape (n_outputs, 2, 2)
        A 2x2 confusion matrix corresponding to each output in the input.
        When calculating class-wise multi_confusion (default), then
        n_outputs = n_labels; when calculating sample-wise multi_confusion
        (samplewise=True), n_outputs = n_samples. If ``labels`` is defined,
        the results will be returned in the order specified in ``labels``,
        otherwise the results will be returned in sorted order by default.

    See Also
    --------
    confusion_matrix

    Notes
    -----
    The multilabel_confusion_matrix calculates class-wise or sample-wise
    multilabel confusion matrices, and in multiclass tasks, labels are
    binarized under a one-vs-rest way; while confusion_matrix calculates
    one confusion matrix for confusion between every two classes.

    Examples
    --------
    Multilabel-indicator case:

    >>> import numpy as np
    >>> from sklearn.metrics import multilabel_confusion_matrix
    >>> y_true = np.array([[1, 0, 1],
    ...                    [0, 1, 0]])
    >>> y_pred = np.array([[1, 0, 0],
    ...                    [0, 1, 1]])
    >>> multilabel_confusion_matrix(y_true, y_pred)
    array([[[1, 0],
            [0, 1]],
    <BLANKLINE>
           [[1, 0],
            [0, 1]],
    <BLANKLINE>
           [[0, 1],
            [1, 0]]])

    Multiclass case:

    >>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
    >>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
    >>> multilabel_confusion_matrix(y_true, y_pred,
```

```
    ...                              labels=["ant", "bird", "cat"])
    array([[[3, 1],
            [0, 2]],
    <BLANKLINE>
           [[5, 0],
            [1, 0]],
    <BLANKLINE>
           [[2, 1],
            [1, 2]]])
    """
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    if sample_weight is not None:
        sample_weight = column_or_1d(sample_weight)
    check_consistent_length(y_true, y_pred, sample_weight)

    if y_type not in ("binary", "multiclass", "multilabel-indicator"):
        raise ValueError("%s is not supported" % y_type)

    present_labels = unique_labels(y_true, y_pred)
    if labels is None:
        labels = present_labels
        n_labels = None
    else:
        n_labels = len(labels)
        labels = np.hstack([labels, np.setdiff1d(present_labels, labels,
                                                 assume_unique=True)])

    if y_true.ndim == 1:
        if samplewise:
            raise ValueError("Samplewise metrics are not available outside of "
                             "multilabel classification.")

        le = LabelEncoder()
        le.fit(labels)
        y_true = le.transform(y_true)
        y_pred = le.transform(y_pred)
        sorted_labels = le.classes_

        # labels are now from 0 to len(labels) - 1 -> use bincount
        tp = y_true == y_pred
        tp_bins = y_true[tp]
        if sample_weight is not None:
            tp_bins_weights = np.asarray(sample_weight)[tp]
        else:
            tp_bins_weights = None

        if len(tp_bins):
            tp_sum = np.bincount(tp_bins, weights=tp_bins_weights,
                                 minlength=len(labels))
        else:
            # Pathological case
            true_sum = pred_sum = tp_sum = np.zeros(len(labels))
```

```
        if len(y_pred):
            pred_sum = np.bincount(y_pred, weights=sample_weight,
                                   minlength=len(labels))
        if len(y_true):
            true_sum = np.bincount(y_true, weights=sample_weight,
                                   minlength=len(labels))

        # Retain only selected labels
        indices = np.searchsorted(sorted_labels, labels[:n_labels])
        tp_sum = tp_sum[indices]
        true_sum = true_sum[indices]
        pred_sum = pred_sum[indices]

    else:
        sum_axis = 1 if samplewise else 0

        # All labels are index integers for multilabel.
        # Select labels:
        if not np.array_equal(labels, present_labels):
            if np.max(labels) > np.max(present_labels):
                raise ValueError('All labels must be in [0, n labels) for '
                                 'multilabel targets. '
                                 'Got %d > %d' %
                                 (np.max(labels), np.max(present_labels)))
            if np.min(labels) < 0:
                raise ValueError('All labels must be in [0, n labels) for '
                                 'multilabel targets. '
                                 'Got %d < 0' % np.min(labels))

        if n_labels is not None:
            y_true = y_true[:, labels[:n_labels]]
            y_pred = y_pred[:, labels[:n_labels]]

        # calculate weighted counts
        true_and_pred = y_true.multiply(y_pred)
        tp_sum = count_nonzero(true_and_pred, axis=sum_axis,
                               sample_weight=sample_weight)
        pred_sum = count_nonzero(y_pred, axis=sum_axis,
                                 sample_weight=sample_weight)
        true_sum = count_nonzero(y_true, axis=sum_axis,
                                 sample_weight=sample_weight)

    fp = pred_sum - tp_sum
    fn = true_sum - tp_sum
    tp = tp_sum

    if sample_weight is not None and samplewise:
        sample_weight = np.array(sample_weight)
        tp = np.array(tp)
        fp = np.array(fp)
        fn = np.array(fn)
        tn = sample_weight * y_true.shape[1] - tp - fp - fn
```

```python
    elif sample_weight is not None:
        tn = sum(sample_weight) - tp - fp - fn
    elif samplewise:
        tn = y_true.shape[1] - tp - fp - fn
    else:
        tn = y_true.shape[0] - tp - fp - fn

    return np.array([tn, fp, fn, tp]).T.reshape(-1, 2, 2)


@_deprecate_positional_args
def cohen_kappa_score(y1, y2, *, labels=None, weights=None,
                      sample_weight=None):
    r"""Cohen's kappa: a statistic that measures inter-annotator agreement.

    This function computes Cohen's kappa [1]_, a score that expresses the level
    of agreement between two annotators on a classification problem. It is
    defined as

    .. math::
        \kappa = (p_o - p_e) / (1 - p_e)

    where :math:`p_o` is the empirical probability of agreement on the label
    assigned to any sample (the observed agreement ratio), and :math:`p_e` is
    the expected agreement when both annotators assign labels randomly.
    :math:`p_e` is estimated using a per-annotator empirical prior over the
    class labels [2]_.

    Read more in the :ref:`User Guide <cohen_kappa>`.

    Parameters
    ----------
    y1 : array of shape (n_samples,)
        Labels assigned by the first annotator.

    y2 : array of shape (n_samples,)
        Labels assigned by the second annotator. The kappa statistic is
        symmetric, so swapping ``y1`` and ``y2`` doesn't change the value.

    labels : array-like of shape (n_classes,), default=None
        List of labels to index the matrix. This may be used to select a
        subset of labels. If None, all labels that appear at least once in
        ``y1`` or ``y2`` are used.

    weights : {'linear', 'quadratic'}, default=None
        Weighting type to calculate the score. None means no weighted;
        "linear" means linear weighted; "quadratic" means quadratic weighted.

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

    Returns
```

```
    -------
    kappa : float
        The kappa statistic, which is a number between -1 and 1. The maximum
        value means complete agreement; zero or lower means chance agreement.

    References
    ----------
    .. [1] J. Cohen (1960). "A coefficient of agreement for nominal scales".
           Educational and Psychological Measurement 20(1):37-46.
           doi:10.1177/001316446002000104.
    .. [2] `R. Artstein and M. Poesio (2008). "Inter-coder agreement for
           computational linguistics". Computational Linguistics 34(4):555-596
           <https://www.mitpressjournals.org/doi/pdf/10.1162/coli.07-034-R2>`_.
    .. [3] `Wikipedia entry for the Cohen's kappa
            <https://en.wikipedia.org/wiki/Cohen%27s_kappa>`_.
    """
    confusion = confusion_matrix(y1, y2, labels=labels,
                                 sample_weight=sample_weight)
    n_classes = confusion.shape[0]
    sum0 = np.sum(confusion, axis=0)
    sum1 = np.sum(confusion, axis=1)
    expected = np.outer(sum0, sum1) / np.sum(sum0)

    if weights is None:
        w_mat = np.ones([n_classes, n_classes], dtype=int)
        w_mat.flat[:: n_classes + 1] = 0
    elif weights == "linear" or weights == "quadratic":
        w_mat = np.zeros([n_classes, n_classes], dtype=int)
        w_mat += np.arange(n_classes)
        if weights == "linear":
            w_mat = np.abs(w_mat - w_mat.T)
        else:
            w_mat = (w_mat - w_mat.T) ** 2
    else:
        raise ValueError("Unknown kappa weighting type.")

    k = np.sum(w_mat * confusion) / np.sum(w_mat * expected)
    return 1 - k


@_deprecate_positional_args
def jaccard_score(y_true, y_pred, *, labels=None, pos_label=1,
                  average='binary', sample_weight=None, zero_division="warn"):
    """Jaccard similarity coefficient score.

    The Jaccard index [1], or Jaccard similarity coefficient, defined as
    the size of the intersection divided by the size of the union of two label
    sets, is used to compare set of predicted labels for a sample to the
    corresponding set of labels in ``y_true``.

    Read more in the :ref:`User Guide <jaccard_similarity_score>`.

```

```
    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

    labels : array-like of shape (n_classes,), default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {None, 'micro', 'macro', 'samples', 'weighted', \
            'binary'}, default='binary'
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.
        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.
        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean.  This does not take label imbalance into account.
        ``'weighted'``:
            Calculate metrics for each label, and find their average, weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance.
        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
            meaningful for multilabel classification).

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

    zero_division : "warn", {0.0, 1.0}, default="warn"
        Sets the value to return when there is a zero division, i.e. when there
        there are no negative values in predictions and labels. If set to
        "warn", this acts like 0, but a warning is also raised.
```

```

    Returns
    -------
    score : float (if average is not None) or array of floats, shape =\
            [n_unique_labels]

    See Also
    --------
    accuracy_score, f_score, multilabel_confusion_matrix

    Notes
    -----
    :func:`jaccard_score` may be a poor metric if there are no
    positives for some samples or classes. Jaccard is undefined if there are
    no true or predicted labels, and our implementation will return a score
    of 0 with a warning.

    References
    ----------
    .. [1] `Wikipedia entry for the Jaccard index
           <https://en.wikipedia.org/wiki/Jaccard_index>`_.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.metrics import jaccard_score
    >>> y_true = np.array([[0, 1, 1],
    ...                    [1, 1, 0]])
    >>> y_pred = np.array([[1, 1, 1],
    ...                    [1, 0, 0]])

    In the binary case:

    >>> jaccard_score(y_true[0], y_pred[0])
    0.6666...

    In the multilabel case:

    >>> jaccard_score(y_true, y_pred, average='samples')
    0.5833...
    >>> jaccard_score(y_true, y_pred, average='macro')
    0.6666...
    >>> jaccard_score(y_true, y_pred, average=None)
    array([0.5, 0.5, 1. ])

    In the multiclass case:

    >>> y_pred = [0, 2, 1, 2]
    >>> y_true = [0, 1, 2, 2]
    >>> jaccard_score(y_true, y_pred, average=None)
    array([1. , 0. , 0.33...])
    """
```

```python
    labels = _check_set_wise_labels(y_true, y_pred, average, labels,
                                    pos_label)
    samplewise = average == 'samples'
    MCM = multilabel_confusion_matrix(y_true, y_pred,
                                      sample_weight=sample_weight,
                                      labels=labels, samplewise=samplewise)
    numerator = MCM[:, 1, 1]
    denominator = MCM[:, 1, 1] + MCM[:, 0, 1] + MCM[:, 1, 0]

    if average == 'micro':
        numerator = np.array([numerator.sum()])
        denominator = np.array([denominator.sum()])

    jaccard = _prf_divide(numerator, denominator, 'jaccard',
                          'true or predicted', average, ('jaccard',),
                          zero_division=zero_division)
    if average is None:
        return jaccard
    if average == 'weighted':
        weights = MCM[:, 1, 0] + MCM[:, 1, 1]
        if not np.any(weights):
            # numerator is 0, and warning should have already been issued
            weights = None
    elif average == 'samples' and sample_weight is not None:
        weights = sample_weight
    else:
        weights = None
    return np.average(jaccard, weights=weights)


@_deprecate_positional_args
def matthews_corrcoef(y_true, y_pred, *, sample_weight=None):
    """Compute the Matthews correlation coefficient (MCC).

    The Matthews correlation coefficient is used in machine learning as a
    measure of the quality of binary and multiclass classifications. It takes
    into account true and false positives and negatives and is generally
    regarded as a balanced measure which can be used even if the classes are of
    very different sizes. The MCC is in essence a correlation coefficient value
    between -1 and +1. A coefficient of +1 represents a perfect prediction, 0
    an average random prediction and -1 an inverse prediction.  The statistic
    is also known as the phi coefficient. [source: Wikipedia]

    Binary and multiclass labels are supported.  Only in the binary case does
    this relate to information about true and false positives and negatives.
    See references below.

    Read more in the :ref:`User Guide <matthews_corrcoef>`.

    Parameters
    ----------
    y_true : array, shape = [n_samples]
```

```
        Ground truth (correct) target values.

    y_pred : array, shape = [n_samples]
        Estimated targets as returned by a classifier.

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

        .. versionadded:: 0.18

    Returns
    -------
    mcc : float
        The Matthews correlation coefficient (+1 represents a perfect
        prediction, 0 an average random prediction and -1 and inverse
        prediction).

    References
    ----------
    .. [1] `Baldi, Brunak, Chauvin, Andersen and Nielsen, (2000). Assessing the
       accuracy of prediction algorithms for classification: an overview
       <https://doi.org/10.1093/bioinformatics/16.5.412>`_.

    .. [2] `Wikipedia entry for the Matthews Correlation Coefficient
       <https://en.wikipedia.org/wiki/Matthews_correlation_coefficient>`_.

    .. [3] `Gorodkin, (2004). Comparing two K-category assignments by a
        K-category correlation coefficient
        <https://www.sciencedirect.com/science/article/pii/S1476927104000799>`_.

    .. [4] `Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN
        Error Measures in MultiClass Prediction
        <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0041882>`_.

    Examples
    --------
    >>> from sklearn.metrics import matthews_corrcoef
    >>> y_true = [+1, +1, +1, -1]
    >>> y_pred = [+1, -1, +1, +1]
    >>> matthews_corrcoef(y_true, y_pred)
    -0.33...
    """
    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    check_consistent_length(y_true, y_pred, sample_weight)
    if y_type not in {"binary", "multiclass"}:
        raise ValueError("%s is not supported" % y_type)

    lb = LabelEncoder()
    lb.fit(np.hstack([y_true, y_pred]))
    y_true = lb.transform(y_true)
    y_pred = lb.transform(y_pred)

```

```
    C = confusion_matrix(y_true, y_pred, sample_weight=sample_weight)
    t_sum = C.sum(axis=1, dtype=np.float64)
    p_sum = C.sum(axis=0, dtype=np.float64)
    n_correct = np.trace(C, dtype=np.float64)
    n_samples = p_sum.sum()
    cov_ytyp = n_correct * n_samples - np.dot(t_sum, p_sum)
    cov_ypyp = n_samples ** 2 - np.dot(p_sum, p_sum)
    cov_ytyt = n_samples ** 2 - np.dot(t_sum, t_sum)
    mcc = cov_ytyp / np.sqrt(cov_ytyt * cov_ypyp)

    if np.isnan(mcc):
        return 0.
    else:
        return mcc


@_deprecate_positional_args
def zero_one_loss(y_true, y_pred, *, normalize=True, sample_weight=None):
    """Zero-one classification loss.

    If normalize is ``True``, return the fraction of misclassifications
    (float), else it returns the number of misclassifications (int). The best
    performance is 0.

    Read more in the :ref:`User Guide <zero_one_loss>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

    normalize : bool, default=True
        If ``False``, return the number of misclassifications.
        Otherwise, return the fraction of misclassifications.

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

    Returns
    -------
    loss : float or int,
        If ``normalize == True``, return the fraction of misclassifications
        (float), else it returns the number of misclassifications (int).

    Notes
    -----
    In multilabel classification, the zero_one_loss function corresponds to
    the subset zero-one loss: for each sample, the entire set of labels must be
    correctly predicted, otherwise the loss for that sample is equal to one.
```

```

    See Also
    --------
    accuracy_score, hamming_loss, jaccard_score

    Examples
    --------
    >>> from sklearn.metrics import zero_one_loss
    >>> y_pred = [1, 2, 3, 4]
    >>> y_true = [2, 2, 3, 4]
    >>> zero_one_loss(y_true, y_pred)
    0.25
    >>> zero_one_loss(y_true, y_pred, normalize=False)
    1

    In the multilabel case with binary label indicators:

    >>> import numpy as np
    >>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
    0.5
    """
    score = accuracy_score(y_true, y_pred,
                           normalize=normalize,
                           sample_weight=sample_weight)

    if normalize:
        return 1 - score
    else:
        if sample_weight is not None:
            n_samples = np.sum(sample_weight)
        else:
            n_samples = _num_samples(y_true)
        return n_samples - score


@_deprecate_positional_args
def f1_score(y_true, y_pred, *, labels=None, pos_label=1, average='binary',
             sample_weight=None, zero_division="warn"):
    """Compute the F1 score, also known as balanced F-score or F-measure.

    The F1 score can be interpreted as a weighted average of the precision and
    recall, where an F1 score reaches its best value at 1 and worst score at 0.
    The relative contribution of precision and recall to the F1 score are
    equal. The formula for the F1 score is::

        F1 = 2 * (precision * recall) / (precision + recall)

    In the multi-class and multi-label case, this is the average of
    the F1 score of each class with weighting depending on the ``average``
    parameter.

    Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.
```

```
    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    labels : array-like, default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

        .. versionchanged:: 0.17
           Parameter `labels` improved for multiclass problem.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {'micro', 'macro', 'samples','weighted', 'binary'} or None, \
            default='binary'
        This parameter is required for multiclass/multilabel targets.
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.
        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.
        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean.  This does not take label imbalance into account.
        ``'weighted'``:
            Calculate metrics for each label, and find their average weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance; it can result in an
            F-score that is not between precision and recall.
        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
            meaningful for multilabel classification where this differs from
            :func:`accuracy_score`).
```

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

    zero_division : "warn", 0 or 1, default="warn"
        Sets the value to return when there is a zero division, i.e. when all
        predictions and labels are negative. If set to "warn", this acts as 0,
        but warnings are also raised.

    Returns
    -------
    f1_score : float or array of float, shape = [n_unique_labels]
        F1 score of the positive class in binary classification or weighted
        average of the F1 scores of each class for the multiclass task.

    See Also
    --------
    fbeta_score, precision_recall_fscore_support, jaccard_score,
    multilabel_confusion_matrix

    References
    ----------
    .. [1] `Wikipedia entry for the F1-score
           <https://en.wikipedia.org/wiki/F1_score>`_.

    Examples
    --------
    >>> from sklearn.metrics import f1_score
    >>> y_true = [0, 1, 2, 0, 1, 2]
    >>> y_pred = [0, 2, 1, 0, 0, 1]
    >>> f1_score(y_true, y_pred, average='macro')
    0.26...
    >>> f1_score(y_true, y_pred, average='micro')
    0.33...
    >>> f1_score(y_true, y_pred, average='weighted')
    0.26...
    >>> f1_score(y_true, y_pred, average=None)
    array([0.8, 0. , 0. ])
    >>> y_true = [0, 0, 0, 0, 0, 0]
    >>> y_pred = [0, 0, 0, 0, 0, 0]
    >>> f1_score(y_true, y_pred, zero_division=1)
    1.0...

    Notes
    -----
    When ``true positive + false positive == 0``, precision is undefined.
    When ``true positive + false negative == 0``, recall is undefined.
    In such cases, by default the metric will be set to 0, as will f-score,
    and ``UndefinedMetricWarning`` will be raised. This behavior can be
    modified with ``zero_division``.
    """
    return fbeta_score(y_true, y_pred, beta=1, labels=labels,
                       pos_label=pos_label, average=average,
```

```python
                            sample_weight=sample_weight,
                            zero_division=zero_division)


@_deprecate_positional_args
def fbeta_score(y_true, y_pred, *, beta, labels=None, pos_label=1,
                average='binary', sample_weight=None, zero_division="warn"):
    """Compute the F-beta score.

    The F-beta score is the weighted harmonic mean of precision and recall,
    reaching its optimal value at 1 and its worst value at 0.

    The `beta` parameter determines the weight of recall in the combined
    score. ``beta < 1`` lends more weight to precision, while ``beta > 1``
    favors recall (``beta -> 0`` considers only precision, ``beta -> +inf``
    only recall).

    Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    beta : float
        Determines the weight of recall in the combined score.

    labels : array-like, default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

        .. versionchanged:: 0.17
           Parameter `labels` improved for multiclass problem.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {'micro', 'macro', 'samples', 'weighted', 'binary'} or None \
            default='binary'
        This parameter is required for multiclass/multilabel targets.
        If ``None``, the scores for each class are returned. Otherwise, this
```

```
    determines the type of averaging performed on the data:

    ``'binary'``:
        Only report results for the class specified by ``pos_label``.
        This is applicable only if targets (``y_{true,pred}``) are binary.
    ``'micro'``:
        Calculate metrics globally by counting the total true positives,
        false negatives and false positives.
    ``'macro'``:
        Calculate metrics for each label, and find their unweighted
        mean.  This does not take label imbalance into account.
    ``'weighted'``:
        Calculate metrics for each label, and find their average weighted
        by support (the number of true instances for each label). This
        alters 'macro' to account for label imbalance; it can result in an
        F-score that is not between precision and recall.
    ``'samples'``:
        Calculate metrics for each instance, and find their average (only
        meaningful for multilabel classification where this differs from
        :func:`accuracy_score`).

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

zero_division : "warn", 0 or 1, default="warn"
    Sets the value to return when there is a zero division, i.e. when all
    predictions and labels are negative. If set to "warn", this acts as 0,
    but warnings are also raised.

Returns
-------
fbeta_score : float (if average is not None) or array of float, shape =\
    [n_unique_labels]
    F-beta score of the positive class in binary classification or weighted
    average of the F-beta score of each class for the multiclass task.

See Also
--------
precision_recall_fscore_support, multilabel_confusion_matrix

Notes
-----
When ``true positive + false positive == 0`` or
``true positive + false negative == 0``, f-score returns 0 and raises
``UndefinedMetricWarning``. This behavior can be
modified with ``zero_division``.

References
----------
.. [1] R. Baeza-Yates and B. Ribeiro-Neto (2011).
       Modern Information Retrieval. Addison Wesley, pp. 327-328.

```

```python
    .. [2] `Wikipedia entry for the F1-score
           <https://en.wikipedia.org/wiki/F1_score>`_.

    Examples
    --------
    >>> from sklearn.metrics import fbeta_score
    >>> y_true = [0, 1, 2, 0, 1, 2]
    >>> y_pred = [0, 2, 1, 0, 0, 1]
    >>> fbeta_score(y_true, y_pred, average='macro', beta=0.5)
    0.23...
    >>> fbeta_score(y_true, y_pred, average='micro', beta=0.5)
    0.33...
    >>> fbeta_score(y_true, y_pred, average='weighted', beta=0.5)
    0.23...
    >>> fbeta_score(y_true, y_pred, average=None, beta=0.5)
    array([0.71..., 0.        , 0.        ])
    """

    _, _, f, _ = precision_recall_fscore_support(y_true, y_pred,
                                                 beta=beta,
                                                 labels=labels,
                                                 pos_label=pos_label,
                                                 average=average,
                                                 warn_for=('f-score',),
                                                 sample_weight=sample_weight,
                                                 zero_division=zero_division)
    return f


def _prf_divide(numerator, denominator, metric,
                modifier, average, warn_for, zero_division="warn"):
    """Performs division and handles divide-by-zero.

    On zero-division, sets the corresponding result elements equal to
    0 or 1 (according to ``zero_division``). Plus, if
    ``zero_division != "warn"`` raises a warning.

    The metric, modifier and average arguments are used only for determining
    an appropriate warning.
    """
    mask = denominator == 0.0
    denominator = denominator.copy()
    denominator[mask] = 1  # avoid infs/nans
    result = numerator / denominator

    if not np.any(mask):
        return result

    # if ``zero_division=1``, set those with denominator == 0 equal to 1
    result[mask] = 0.0 if zero_division in ["warn", 0] else 1.0

    # the user will be removing warnings if zero_division is set to something
```

```python
    # different than its default value. If we are computing only f-score
    # the warning will be raised only if precision and recall are ill-defined
    if zero_division != "warn" or metric not in warn_for:
        return result

    # build appropriate warning
    # E.g. "Precision and F-score are ill-defined and being set to 0.0 in
    # labels with no predicted samples. Use ``zero_division`` parameter to
    # control this behavior."

    if metric in warn_for and 'f-score' in warn_for:
        msg_start = '{0} and F-score are'.format(metric.title())
    elif metric in warn_for:
        msg_start = '{0} is'.format(metric.title())
    elif 'f-score' in warn_for:
        msg_start = 'F-score is'
    else:
        return result

    _warn_prf(average, modifier, msg_start, len(result))

    return result


def _warn_prf(average, modifier, msg_start, result_size):
    axis0, axis1 = 'sample', 'label'
    if average == 'samples':
        axis0, axis1 = axis1, axis0
    msg = ('{0} ill-defined and being set to 0.0 {{0}} '
           'no {1} {2}s. Use `zero_division` parameter to control'
           ' this behavior.'.format(msg_start, modifier, axis0))
    if result_size == 1:
        msg = msg.format('due to')
    else:
        msg = msg.format('in {0}s with'.format(axis1))
    warnings.warn(msg, UndefinedMetricWarning, stacklevel=2)


def _check_set_wise_labels(y_true, y_pred, average, labels, pos_label):
    """Validation associated with set-wise metrics.

    Returns identified labels.
    """
    average_options = (None, 'micro', 'macro', 'weighted', 'samples')
    if average not in average_options and average != 'binary':
        raise ValueError('average has to be one of ' +
                         str(average_options))

    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    # Convert to Python primitive type to avoid NumPy type / Python str
    # comparison. See https://github.com/numpy/numpy/issues/6784
    present_labels = unique_labels(y_true, y_pred).tolist()
```

```python
    if average == 'binary':
        if y_type == 'binary':
            if pos_label not in present_labels:
                if len(present_labels) >= 2:
                    raise ValueError(
                        f"pos_label={pos_label} is not a valid label. It "
                        f"should be one of {present_labels}"
                    )
            labels = [pos_label]
        else:
            average_options = list(average_options)
            if y_type == 'multiclass':
                average_options.remove('samples')
            raise ValueError("Target is %s but average='binary'. Please "
                             "choose another average setting, one of %r."
                             % (y_type, average_options))
    elif pos_label not in (None, 1):
        warnings.warn("Note that pos_label (set to %r) is ignored when "
                      "average != 'binary' (got %r). You may use "
                      "labels=[pos_label] to specify a single positive class."
                      % (pos_label, average), UserWarning)
    return labels


@_deprecate_positional_args
def precision_recall_fscore_support(y_true, y_pred, *, beta=1.0, labels=None,
                                    pos_label=1, average=None,
                                    warn_for=('precision', 'recall',
                                              'f-score'),
                                    sample_weight=None,
                                    zero_division="warn"):
    """Compute precision, recall, F-measure and support for each class.

    The precision is the ratio ``tp / (tp + fp)`` where ``tp`` is the number of
    true positives and ``fp`` the number of false positives. The precision is
    intuitively the ability of the classifier not to label as positive a sample
    that is negative.

    The recall is the ratio ``tp / (tp + fn)`` where ``tp`` is the number of
    true positives and ``fn`` the number of false negatives. The recall is
    intuitively the ability of the classifier to find all the positive samples.

    The F-beta score can be interpreted as a weighted harmonic mean of
    the precision and recall, where an F-beta score reaches its best
    value at 1 and worst score at 0.

    The F-beta score weights recall more than precision by a factor of
    ``beta``. ``beta == 1.0`` means recall and precision are equally important.

    The support is the number of occurrences of each class in ``y_true``.

    If ``pos_label is None`` and in binary classification, this function
```

```
    returns the average precision, recall and F-measure if ``average``
    is one of ``'micro'``, ``'macro'``, ``'weighted'`` or ``'samples'``.

    Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    beta : float, default=1.0
        The strength of recall versus precision in the F-score.

    labels : array-like, default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {'binary', 'micro', 'macro', 'samples','weighted'}, \
            default=None
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.
        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.
        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean.  This does not take label imbalance into account.
        ``'weighted'``:
            Calculate metrics for each label, and find their average weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance; it can result in an
            F-score that is not between precision and recall.
        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
```

```
            meaningful for multilabel classification where this differs from
            :func:`accuracy_score`).

    warn_for : tuple or set, for internal use
        This determines which warnings will be made in the case that this
        function is being used to return only one of its metrics.

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

    zero_division : "warn", 0 or 1, default="warn"
        Sets the value to return when there is a zero division:
           - recall: when there are no positive labels
           - precision: when there are no positive predictions
           - f-score: both

        If set to "warn", this acts as 0, but warnings are also raised.

    Returns
    -------
    precision : float (if average is not None) or array of float, shape =\
        [n_unique_labels]

    recall : float (if average is not None) or array of float, , shape =\
        [n_unique_labels]

    fbeta_score : float (if average is not None) or array of float, shape =\
        [n_unique_labels]

    support : None (if average is not None) or array of int, shape =\
        [n_unique_labels]
        The number of occurrences of each label in ``y_true``.

    Notes
    -----
    When ``true positive + false positive == 0``, precision is undefined.
    When ``true positive + false negative == 0``, recall is undefined.
    In such cases, by default the metric will be set to 0, as will f-score,
    and ``UndefinedMetricWarning`` will be raised. This behavior can be
    modified with ``zero_division``.

    References
    ----------
    .. [1] `Wikipedia entry for the Precision and recall
           <https://en.wikipedia.org/wiki/Precision_and_recall>`_.

    .. [2] `Wikipedia entry for the F1-score
           <https://en.wikipedia.org/wiki/F1_score>`_.

    .. [3] `Discriminative Methods for Multi-labeled Classification Advances
           in Knowledge Discovery and Data Mining (2004), pp. 22-30 by Shantanu
           Godbole, Sunita Sarawagi
```

```
        <http://www.godbole.net/shantanu/pubs/multilabelsvm-pakdd04.pdf>`_.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.metrics import precision_recall_fscore_support
    >>> y_true = np.array(['cat', 'dog', 'pig', 'cat', 'dog', 'pig'])
    >>> y_pred = np.array(['cat', 'pig', 'dog', 'cat', 'cat', 'dog'])
    >>> precision_recall_fscore_support(y_true, y_pred, average='macro')
    (0.22..., 0.33..., 0.26..., None)
    >>> precision_recall_fscore_support(y_true, y_pred, average='micro')
    (0.33..., 0.33..., 0.33..., None)
    >>> precision_recall_fscore_support(y_true, y_pred, average='weighted')
    (0.22..., 0.33..., 0.26..., None)

    It is possible to compute per-label precisions, recalls, F1-scores and
    supports instead of averaging:

    >>> precision_recall_fscore_support(y_true, y_pred, average=None,
    ... labels=['pig', 'dog', 'cat'])
    (array([0.        , 0.        , 0.66...]),
     array([0., 0., 1.]), array([0. , 0. , 0.8]),
     array([2, 2, 2]))
    """
    _check_zero_division(zero_division)
    if beta < 0:
        raise ValueError("beta should be >=0 in the F-beta score")
    labels = _check_set_wise_labels(y_true, y_pred, average, labels,
                                    pos_label)

    # Calculate tp_sum, pred_sum, true_sum ###
    samplewise = average == 'samples'
    MCM = multilabel_confusion_matrix(y_true, y_pred,
                                      sample_weight=sample_weight,
                                      labels=labels, samplewise=samplewise)
    tp_sum = MCM[:, 1, 1]
    pred_sum = tp_sum + MCM[:, 0, 1]
    true_sum = tp_sum + MCM[:, 1, 0]

    if average == 'micro':
        tp_sum = np.array([tp_sum.sum()])
        pred_sum = np.array([pred_sum.sum()])
        true_sum = np.array([true_sum.sum()])

    # Finally, we have all our sufficient statistics. Divide! #
    beta2 = beta ** 2

    # Divide, and on zero-division, set scores and/or warn according to
    # zero_division:
    precision = _prf_divide(tp_sum, pred_sum, 'precision',
                            'predicted', average, warn_for, zero_division)
    recall = _prf_divide(tp_sum, true_sum, 'recall',
```

```
                                 'true', average, warn_for, zero_division)

        # warn for f-score only if zero_division is warn, it is in warn_for
        # and BOTH prec and rec are ill-defined
        if zero_division == "warn" and ("f-score",) == warn_for:
            if (pred_sum[true_sum == 0] == 0).any():
                _warn_prf(
                    average, "true nor predicted", 'F-score is', len(true_sum)
                )

        # if tp == 0 F will be 1 only if all predictions are zero, all labels are
        # zero, and zero_division=1. In all other case, 0
        if np.isposinf(beta):
            f_score = recall
        else:
            denom = beta2 * precision + recall

            denom[denom == 0.] = 1  # avoid division by 0
            f_score = (1 + beta2) * precision * recall / denom

        # Average the results
        if average == 'weighted':
            weights = true_sum
            if weights.sum() == 0:
                zero_division_value = np.float64(1.0)
                if zero_division in ["warn", 0]:
                    zero_division_value = np.float64(0.0)
                # precision is zero_division if there are no positive predictions
                # recall is zero_division if there are no positive labels
                # fscore is zero_division if all labels AND predictions are
                # negative
                if pred_sum.sum() == 0:
                    return (zero_division_value,
                            zero_division_value,
                            zero_division_value,
                            None)
                else:
                    return (np.float64(0.0),
                            zero_division_value,
                            np.float64(0.0),
                            None)

        elif average == 'samples':
            weights = sample_weight
        else:
            weights = None

        if average is not None:
            assert average != 'binary' or len(precision) == 1
            precision = np.average(precision, weights=weights)
            recall = np.average(recall, weights=weights)
            f_score = np.average(f_score, weights=weights)
```

```
            true_sum = None  # return no support

    return precision, recall, f_score, true_sum


@_deprecate_positional_args
def precision_score(y_true, y_pred, *, labels=None, pos_label=1,
                    average='binary', sample_weight=None,
                    zero_division="warn"):
    """Compute the precision.

    The precision is the ratio ``tp / (tp + fp)`` where ``tp`` is the number of
    true positives and ``fp`` the number of false positives. The precision is
    intuitively the ability of the classifier not to label as positive a sample
    that is negative.

    The best value is 1 and the worst value is 0.

    Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    labels : array-like, default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

        .. versionchanged:: 0.17
           Parameter `labels` improved for multiclass problem.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {'micro', 'macro', 'samples', 'weighted', 'binary'} \
            default='binary'
        This parameter is required for multiclass/multilabel targets.
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

```

```
        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.
        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.
        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean.  This does not take label imbalance into account.
        ``'weighted'``:
            Calculate metrics for each label, and find their average weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance; it can result in an
            F-score that is not between precision and recall.
        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
            meaningful for multilabel classification where this differs from
            :func:`accuracy_score`).

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

    zero_division : "warn", 0 or 1, default="warn"
        Sets the value to return when there is a zero division. If set to
        "warn", this acts as 0, but warnings are also raised.

    Returns
    -------
    precision : float (if average is not None) or array of float of shape
        (n_unique_labels,)
        Precision of the positive class in binary classification or weighted
        average of the precision of each class for the multiclass task.

    See Also
    --------
    precision_recall_fscore_support, multilabel_confusion_matrix

    Notes
    -----
    When ``true positive + false positive == 0``, precision returns 0 and
    raises ``UndefinedMetricWarning``. This behavior can be
    modified with ``zero_division``.

    Examples
    --------
    >>> from sklearn.metrics import precision_score
    >>> y_true = [0, 1, 2, 0, 1, 2]
    >>> y_pred = [0, 2, 1, 0, 0, 1]
    >>> precision_score(y_true, y_pred, average='macro')
    0.22...
    >>> precision_score(y_true, y_pred, average='micro')
    0.33...
```

```
    >>> precision_score(y_true, y_pred, average='weighted')
    0.22...
    >>> precision_score(y_true, y_pred, average=None)
    array([0.66..., 0.        , 0.        ])
    >>> y_pred = [0, 0, 0, 0, 0, 0]
    >>> precision_score(y_true, y_pred, average=None)
    array([0.33..., 0.        , 0.        ])
    >>> precision_score(y_true, y_pred, average=None, zero_division=1)
    array([0.33..., 1.        , 1.        ])

    """
    p, _, _, _ = precision_recall_fscore_support(y_true, y_pred,
                                                 labels=labels,
                                                 pos_label=pos_label,
                                                 average=average,
                                                 warn_for=('precision',),
                                                 sample_weight=sample_weight,
                                                 zero_division=zero_division)
    return p


@_deprecate_positional_args
def recall_score(y_true, y_pred, *, labels=None, pos_label=1, average='binary',
                 sample_weight=None, zero_division="warn"):
    """Compute the recall.

    The recall is the ratio ``tp / (tp + fn)`` where ``tp`` is the number of
    true positives and ``fn`` the number of false negatives. The recall is
    intuitively the ability of the classifier to find all the positive samples.

    The best value is 1 and the worst value is 0.

    Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    labels : array-like, default=None
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

        .. versionchanged:: 0.17
```

```
            Parameter `labels` improved for multiclass problem.

    pos_label : str or int, default=1
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : {'micro', 'macro', 'samples', 'weighted', 'binary'} \
            default='binary'
        This parameter is required for multiclass/multilabel targets.
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.
        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.
        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean.  This does not take label imbalance into account.
        ``'weighted'``:
            Calculate metrics for each label, and find their average weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance; it can result in an
            F-score that is not between precision and recall.
        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
            meaningful for multilabel classification where this differs from
            :func:`accuracy_score`).

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

    zero_division : "warn", 0 or 1, default="warn"
        Sets the value to return when there is a zero division. If set to
        "warn", this acts as 0, but warnings are also raised.

    Returns
    -------
    recall : float (if average is not None) or array of float of shape
        (n_unique_labels,)
        Recall of the positive class in binary classification or weighted
        average of the recall of each class for the multiclass task.

    See Also
    --------
    precision_recall_fscore_support, balanced_accuracy_score,
    multilabel_confusion_matrix

```

```
    Notes
    -----
    When ``true positive + false negative == 0``, recall returns 0 and raises
    ``UndefinedMetricWarning``. This behavior can be modified with
    ``zero_division``.

    Examples
    --------
    >>> from sklearn.metrics import recall_score
    >>> y_true = [0, 1, 2, 0, 1, 2]
    >>> y_pred = [0, 2, 1, 0, 0, 1]
    >>> recall_score(y_true, y_pred, average='macro')
    0.33...
    >>> recall_score(y_true, y_pred, average='micro')
    0.33...
    >>> recall_score(y_true, y_pred, average='weighted')
    0.33...
    >>> recall_score(y_true, y_pred, average=None)
    array([1., 0., 0.])
    >>> y_true = [0, 0, 0, 0, 0, 0]
    >>> recall_score(y_true, y_pred, average=None)
    array([0.5, 0. , 0. ])
    >>> recall_score(y_true, y_pred, average=None, zero_division=1)
    array([0.5, 1. , 1. ])
    """
    _, r, _, _ = precision_recall_fscore_support(y_true, y_pred,
                                                 labels=labels,
                                                 pos_label=pos_label,
                                                 average=average,
                                                 warn_for=('recall',),
                                                 sample_weight=sample_weight,
                                                 zero_division=zero_division)
    return r


@_deprecate_positional_args
def balanced_accuracy_score(y_true, y_pred, *, sample_weight=None,
                            adjusted=False):
    """Compute the balanced accuracy.

    The balanced accuracy in binary and multiclass classification problems to
    deal with imbalanced datasets. It is defined as the average of recall
    obtained on each class.

    The best value is 1 and the worst value is 0 when ``adjusted=False``.

    Read more in the :ref:`User Guide <balanced_accuracy_score>`.

    .. versionadded:: 0.20

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

```
    y_true : 1d array-like
        Ground truth (correct) target values.

    y_pred : 1d array-like
        Estimated targets as returned by a classifier.

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

    adjusted : bool, default=False
        When true, the result is adjusted for chance, so that random
        performance would score 0, while keeping perfect performance at a score
        of 1.

    Returns
    -------
    balanced_accuracy : float

    See Also
    --------
    recall_score, roc_auc_score

    Notes
    -----
    Some literature promotes alternative definitions of balanced accuracy. Our
    definition is equivalent to :func:`accuracy_score` with class-balanced
    sample weights, and shares desirable properties with the binary case.
    See the :ref:`User Guide <balanced_accuracy_score>`.

    References
    ----------
    .. [1] Brodersen, K.H.; Ong, C.S.; Stephan, K.E.; Buhmann, J.M. (2010).
           The balanced accuracy and its posterior distribution.
           Proceedings of the 20th International Conference on Pattern
           Recognition, 3121-24.
    .. [2] John. D. Kelleher, Brian Mac Namee, Aoife D'Arcy, (2015).
           `Fundamentals of Machine Learning for Predictive Data Analytics:
           Algorithms, Worked Examples, and Case Studies
           <https://mitpress.mit.edu/books/fundamentals-machine-learning-predictive-data-analyt

    Examples
    --------
    >>> from sklearn.metrics import balanced_accuracy_score
    >>> y_true = [0, 1, 0, 0, 1, 0]
    >>> y_pred = [0, 1, 0, 0, 0, 1]
    >>> balanced_accuracy_score(y_true, y_pred)
    0.625

    """
    C = confusion_matrix(y_true, y_pred, sample_weight=sample_weight)
    with np.errstate(divide='ignore', invalid='ignore'):
        per_class = np.diag(C) / C.sum(axis=1)
```

```
    if np.any(np.isnan(per_class)):
        warnings.warn('y_pred contains classes not in y_true')
        per_class = per_class[~np.isnan(per_class)]
    score = np.mean(per_class)
    if adjusted:
        n_classes = len(per_class)
        chance = 1 / n_classes
        score -= chance
        score /= 1 - chance
    return score


@_deprecate_positional_args
def classification_report(y_true, y_pred, *, labels=None, target_names=None,
                          sample_weight=None, digits=2, output_dict=False,
                          zero_division="warn"):
    """Build a text report showing the main classification metrics.

    Read more in the :ref:`User Guide <classification_report>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) target values.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Estimated targets as returned by a classifier.

    labels : array-like of shape (n_labels,), default=None
        Optional list of label indices to include in the report.

    target_names : list of str of shape (n_labels,), default=None
        Optional display names matching the labels (same order).

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

    digits : int, default=2
        Number of digits for formatting output floating point values.
        When ``output_dict`` is ``True``, this will be ignored and the
        returned values will not be rounded.

    output_dict : bool, default=False
        If True, return output as dict.

        .. versionadded:: 0.20

    zero_division : "warn", 0 or 1, default="warn"
        Sets the value to return when there is a zero division. If set to
        "warn", this acts as 0, but warnings are also raised.

    Returns
```

```
-------
report : string / dict
    Text summary of the precision, recall, F1 score for each class.
    Dictionary returned if output_dict is True. Dictionary has the
    following structure::

        {'label 1': {'precision':0.5,
                     'recall':1.0,
                     'f1-score':0.67,
                     'support':1},
         'label 2': { ... },
          ...
        }

    The reported averages include macro average (averaging the unweighted
    mean per label), weighted average (averaging the support-weighted mean
    per label), and sample average (only for multilabel classification).
    Micro average (averaging the total true positives, false negatives and
    false positives) is only shown for multi-label or multi-class
    with a subset of classes, because it corresponds to accuracy
    otherwise and would be the same for all metrics.
    See also :func:`precision_recall_fscore_support` for more details
    on averages.

    Note that in binary classification, recall of the positive class
    is also known as "sensitivity"; recall of the negative class is
    "specificity".

See Also
--------
precision_recall_fscore_support, confusion_matrix,
multilabel_confusion_matrix

Examples
--------
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 2]
>>> y_pred = [0, 0, 2, 2, 1]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
              precision    recall  f1-score   support
<BLANKLINE>
     class 0       0.50      1.00      0.67         1
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.67      0.80         3
<BLANKLINE>
    accuracy                           0.60         5
   macro avg       0.50      0.56      0.49         5
weighted avg       0.70      0.60      0.61         5
<BLANKLINE>
>>> y_pred = [1, 1, 0]
>>> y_true = [1, 1, 1]
```

```
    >>> print(classification_report(y_true, y_pred, labels=[1, 2, 3]))
                  precision    recall  f1-score   support
    <BLANKLINE>
               1       1.00      0.67      0.80         3
               2       0.00      0.00      0.00         0
               3       0.00      0.00      0.00         0
    <BLANKLINE>
       micro avg       1.00      0.67      0.80         3
       macro avg       0.33      0.22      0.27         3
    weighted avg       1.00      0.67      0.80         3
    <BLANKLINE>
    """

    y_type, y_true, y_pred = _check_targets(y_true, y_pred)

    if labels is None:
        labels = unique_labels(y_true, y_pred)
        labels_given = False
    else:
        labels = np.asarray(labels)
        labels_given = True

    # labelled micro average
    micro_is_accuracy = ((y_type == 'multiclass' or y_type == 'binary') and
                         (not labels_given or
                          (set(labels) == set(unique_labels(y_true, y_pred)))))

    if target_names is not None and len(labels) != len(target_names):
        if labels_given:
            warnings.warn(
                "labels size, {0}, does not match size of target_names, {1}"
                .format(len(labels), len(target_names))
            )
        else:
            raise ValueError(
                "Number of classes, {0}, does not match size of "
                "target_names, {1}. Try specifying the labels "
                "parameter".format(len(labels), len(target_names))
            )
    if target_names is None:
        target_names = ['%s' % l for l in labels]

    headers = ["precision", "recall", "f1-score", "support"]
    # compute per-class results without averaging
    p, r, f1, s = precision_recall_fscore_support(y_true, y_pred,
                                                  labels=labels,
                                                  average=None,
                                                  sample_weight=sample_weight,
                                                  zero_division=zero_division)
    rows = zip(target_names, p, r, f1, s)

    if y_type.startswith('multilabel'):
```

```
        average_options = ('micro', 'macro', 'weighted', 'samples')
    else:
        average_options = ('micro', 'macro', 'weighted')

    if output_dict:
        report_dict = {label[0]: label[1:] for label in rows}
        for label, scores in report_dict.items():
            report_dict[label] = dict(zip(headers,
                                          [i.item() for i in scores]))
    else:
        longest_last_line_heading = 'weighted avg'
        name_width = max(len(cn) for cn in target_names)
        width = max(name_width, len(longest_last_line_heading), digits)
        head_fmt = '{:>{width}s} ' + ' {:>9}' * len(headers)
        report = head_fmt.format('', *headers, width=width)
        report += '\n\n'
        row_fmt = '{:>{width}s} ' + ' {:>9.{digits}f}' * 3 + ' {:>9}\n'
        for row in rows:
            report += row_fmt.format(*row, width=width, digits=digits)
        report += '\n'

    # compute all applicable averages
    for average in average_options:
        if average.startswith('micro') and micro_is_accuracy:
            line_heading = 'accuracy'
        else:
            line_heading = average + ' avg'

        # compute averages with specified averaging method
        avg_p, avg_r, avg_f1, _ = precision_recall_fscore_support(
            y_true, y_pred, labels=labels,
            average=average, sample_weight=sample_weight,
            zero_division=zero_division)
        avg = [avg_p, avg_r, avg_f1, np.sum(s)]

        if output_dict:
            report_dict[line_heading] = dict(
                zip(headers, [i.item() for i in avg]))
        else:
            if line_heading == 'accuracy':
                row_fmt_accuracy = '{:>{width}s} ' + \
                        ' {:>9.{digits}}' * 2 + ' {:>9.{digits}f}' + \
                        ' {:>9}\n'
                report += row_fmt_accuracy.format(line_heading, '', '',
                                                  *avg[2:], width=width,
                                                  digits=digits)
            else:
                report += row_fmt.format(line_heading, *avg,
                                         width=width, digits=digits)

    if output_dict:
        if 'accuracy' in report_dict.keys():
```

```
            report_dict['accuracy'] = report_dict['accuracy']['precision']
        return report_dict
    else:
        return report


@_deprecate_positional_args
def hamming_loss(y_true, y_pred, *, sample_weight=None):
    """Compute the average Hamming loss.

    The Hamming loss is the fraction of labels that are incorrectly predicted.

    Read more in the :ref:`User Guide <hamming_loss>`.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

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

        .. versionadded:: 0.18

    Returns
    -------
    loss : float or int
        Return the average Hamming loss between element of ``y_true`` and
        ``y_pred``.

    See Also
    --------
    accuracy_score, jaccard_score, zero_one_loss

    Notes
    -----
    In multiclass classification, the Hamming loss corresponds to the Hamming
    distance between ``y_true`` and ``y_pred`` which is equivalent to the
    subset ``zero_one_loss`` function, when `normalize` parameter is set to
    True.

    In multilabel classification, the Hamming loss is different from the
    subset zero-one loss. The zero-one loss considers the entire set of labels
    for a given sample incorrect if it does not entirely match the true set of
    labels. Hamming loss is more forgiving in that it penalizes only the
    individual labels.

    The Hamming loss is upperbounded by the subset zero-one loss, when
    `normalize` parameter is set to True. It is always between 0 and 1,
```

```
    lower being better.

    References
    ----------
    .. [1] Grigorios Tsoumakas, Ioannis Katakis. Multi-Label Classification:
           An Overview. International Journal of Data Warehousing & Mining,
           3(3), 1-13, July-September 2007.

    .. [2] `Wikipedia entry on the Hamming distance
           <https://en.wikipedia.org/wiki/Hamming_distance>`_.

    Examples
    --------
    >>> from sklearn.metrics import hamming_loss
    >>> y_pred = [1, 2, 3, 4]
    >>> y_true = [2, 2, 3, 4]
    >>> hamming_loss(y_true, y_pred)
    0.25

    In the multilabel case with binary label indicators:

    >>> import numpy as np
    >>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
    0.75
    """

    y_type, y_true, y_pred = _check_targets(y_true, y_pred)
    check_consistent_length(y_true, y_pred, sample_weight)

    if sample_weight is None:
        weight_average = 1.
    else:
        weight_average = np.mean(sample_weight)

    if y_type.startswith('multilabel'):
        n_differences = count_nonzero(y_true - y_pred,
                                      sample_weight=sample_weight)
        return (n_differences /
                (y_true.shape[0] * y_true.shape[1] * weight_average))

    elif y_type in ["binary", "multiclass"]:
        return _weighted_sum(y_true != y_pred, sample_weight, normalize=True)
    else:
        raise ValueError("{0} is not supported".format(y_type))


@_deprecate_positional_args
def log_loss(y_true, y_pred, *, eps=1e-15, normalize=True, sample_weight=None,
             labels=None):
    r"""Log loss, aka logistic loss or cross-entropy loss.

    This is the loss function used in (multinomial) logistic regression
```

```
    and extensions of it such as neural networks, defined as the negative
    log-likelihood of a logistic model that returns ``y_pred`` probabilities
    for its training data ``y_true``.
    The log loss is only defined for two or more labels.
    For a single sample with true label :math:`y \in \{0,1\}` and
    and a probability estimate :math:`p = \operatorname{Pr}(y = 1)`, the log
    loss is:

    .. math::
        L_{\log}(y, p) = -(y \log (p) + (1 - y) \log (1 - p))

    Read more in the :ref:`User Guide <log_loss>`.

    Parameters
    ----------
    y_true : array-like or label indicator matrix
        Ground truth (correct) labels for n_samples samples.

    y_pred : array-like of float, shape = (n_samples, n_classes) or (n_samples,)
        Predicted probabilities, as returned by a classifier's
        predict_proba method. If ``y_pred.shape = (n_samples,)``
        the probabilities provided are assumed to be that of the
        positive class. The labels in ``y_pred`` are assumed to be
        ordered alphabetically, as done by
        :class:`preprocessing.LabelBinarizer`.

    eps : float, default=1e-15
        Log loss is undefined for p=0 or p=1, so probabilities are
        clipped to max(eps, min(1 - eps, p)).

    normalize : bool, default=True
        If true, return the mean loss per sample.
        Otherwise, return the sum of the per-sample losses.

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

    labels : array-like, default=None
        If not provided, labels will be inferred from y_true. If ``labels``
        is ``None`` and ``y_pred`` has shape (n_samples,) the labels are
        assumed to be binary and are inferred from ``y_true``.

        .. versionadded:: 0.18

    Returns
    -------
    loss : float

    Notes
    -----
    The logarithm used is the natural logarithm (base-e).

```

```
    Examples
    --------
    >>> from sklearn.metrics import log_loss
    >>> log_loss(["spam", "ham", "ham", "spam"],
    ...          [[.1, .9], [.9, .1], [.8, .2], [.35, .65]])
    0.21616...

    References
    ----------
    C.M. Bishop (2006). Pattern Recognition and Machine Learning. Springer,
    p. 209.
    """
    y_pred = check_array(y_pred, ensure_2d=False)
    check_consistent_length(y_pred, y_true, sample_weight)

    lb = LabelBinarizer()

    if labels is not None:
        lb.fit(labels)
    else:
        lb.fit(y_true)

    if len(lb.classes_) == 1:
        if labels is None:
            raise ValueError('y_true contains only one label ({0}). Please '
                             'provide the true labels explicitly through the '
                             'labels argument.'.format(lb.classes_[0]))
        else:
            raise ValueError('The labels array needs to contain at least two '
                             'labels for log_loss, '
                             'got {0}.'.format(lb.classes_))

    transformed_labels = lb.transform(y_true)

    if transformed_labels.shape[1] == 1:
        transformed_labels = np.append(1 - transformed_labels,
                                       transformed_labels, axis=1)

    # Clipping
    y_pred = np.clip(y_pred, eps, 1 - eps)

    # If y_pred is of single dimension, assume y_true to be binary
    # and then check.
    if y_pred.ndim == 1:
        y_pred = y_pred[:, np.newaxis]
    if y_pred.shape[1] == 1:
        y_pred = np.append(1 - y_pred, y_pred, axis=1)

    # Check if dimensions are consistent.
    transformed_labels = check_array(transformed_labels)
    if len(lb.classes_) != y_pred.shape[1]:
        if labels is None:
```

```
            raise ValueError("y_true and y_pred contain different number of "
                             "classes {0}, {1}. Please provide the true "
                             "labels explicitly through the labels argument. "
                             "Classes found in "
                             "y_true: {2}".format(transformed_labels.shape[1],
                                                  y_pred.shape[1],
                                                  lb.classes_))
        else:
            raise ValueError('The number of classes in labels is different '
                             'from that in y_pred. Classes found in '
                             'labels: {0}'.format(lb.classes_))

    # Renormalize
    y_pred /= y_pred.sum(axis=1)[:, np.newaxis]
    loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)

    return _weighted_sum(loss, sample_weight, normalize)


@_deprecate_positional_args
def hinge_loss(y_true, pred_decision, *, labels=None, sample_weight=None):
    """Average hinge loss (non-regularized).

    In binary class case, assuming labels in y_true are encoded with +1 and -1,
    when a prediction mistake is made, ``margin = y_true * pred_decision`` is
    always negative (since the signs disagree), implying ``1 - margin`` is
    always greater than 1.  The cumulated hinge loss is therefore an upper
    bound of the number of mistakes made by the classifier.

    In multiclass case, the function expects that either all the labels are
    included in y_true or an optional labels argument is provided which
    contains all the labels. The multilabel margin is calculated according
    to Crammer-Singer's method. As in the binary case, the cumulated hinge loss
    is an upper bound of the number of mistakes made by the classifier.

    Read more in the :ref:`User Guide <hinge_loss>`.

    Parameters
    ----------
    y_true : array of shape (n_samples,)
        True target, consisting of integers of two values. The positive label
        must be greater than the negative label.

    pred_decision : array of shape (n_samples,) or (n_samples, n_classes)
        Predicted decisions, as output by decision_function (floats).

    labels : array-like, default=None
        Contains all the labels for the problem. Used in multiclass hinge loss.

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

```

```
    Returns
    -------
    loss : float

    References
    ----------
    .. [1] `Wikipedia entry on the Hinge loss
           <https://en.wikipedia.org/wiki/Hinge_loss>`_.

    .. [2] Koby Crammer, Yoram Singer. On the Algorithmic
           Implementation of Multiclass Kernel-based Vector
           Machines. Journal of Machine Learning Research 2,
           (2001), 265-292.

    .. [3] `L1 AND L2 Regularization for Multiclass Hinge Loss Models
           by Robert C. Moore, John DeNero
           <http://www.ttic.edu/sigml/symposium2011/papers/
           Moore+DeNero_Regularization.pdf>`_.

    Examples
    --------
    >>> from sklearn import svm
    >>> from sklearn.metrics import hinge_loss
    >>> X = [[0], [1]]
    >>> y = [-1, 1]
    >>> est = svm.LinearSVC(random_state=0)
    >>> est.fit(X, y)
    LinearSVC(random_state=0)
    >>> pred_decision = est.decision_function([[-2], [3], [0.5]])
    >>> pred_decision
    array([-2.18...,  2.36...,  0.09...])
    >>> hinge_loss([-1, 1, 1], pred_decision)
    0.30...

    In the multiclass case:

    >>> import numpy as np
    >>> X = np.array([[0], [1], [2], [3]])
    >>> Y = np.array([0, 1, 2, 3])
    >>> labels = np.array([0, 1, 2, 3])
    >>> est = svm.LinearSVC()
    >>> est.fit(X, Y)
    LinearSVC()
    >>> pred_decision = est.decision_function([[-1], [2], [3]])
    >>> y_true = [0, 2, 3]
    >>> hinge_loss(y_true, pred_decision, labels=labels)
    0.56...
    """
    check_consistent_length(y_true, pred_decision, sample_weight)
    pred_decision = check_array(pred_decision, ensure_2d=False)
    y_true = column_or_1d(y_true)
    y_true_unique = np.unique(labels if labels is not None else y_true)
```

```
    if y_true_unique.size > 2:
        if (labels is None and pred_decision.ndim > 1 and
                (np.size(y_true_unique) != pred_decision.shape[1])):
            raise ValueError("Please include all labels in y_true "
                             "or pass labels as third argument")
        if labels is None:
            labels = y_true_unique
        le = LabelEncoder()
        le.fit(labels)
        y_true = le.transform(y_true)
        mask = np.ones_like(pred_decision, dtype=bool)
        mask[np.arange(y_true.shape[0]), y_true] = False
        margin = pred_decision[~mask]
        margin -= np.max(pred_decision[mask].reshape(y_true.shape[0], -1),
                         axis=1)

    else:
        # Handles binary class case
        # this code assumes that positive and negative labels
        # are encoded as +1 and -1 respectively
        pred_decision = column_or_1d(pred_decision)
        pred_decision = np.ravel(pred_decision)

        lbin = LabelBinarizer(neg_label=-1)
        y_true = lbin.fit_transform(y_true)[:, 0]

        try:
            margin = y_true * pred_decision
        except TypeError:
            raise TypeError("pred_decision should be an array of floats.")

    losses = 1 - margin
    # The hinge_loss doesn't penalize good enough predictions.
    np.clip(losses, 0, None, out=losses)
    return np.average(losses, weights=sample_weight)


@_deprecate_positional_args
def brier_score_loss(y_true, y_prob, *, sample_weight=None, pos_label=None):
    """Compute the Brier score loss.

    The smaller the Brier score loss, the better, hence the naming with "loss".
    The Brier score measures the mean squared difference between the predicted
    probability and the actual outcome. The Brier score always
    takes on a value between zero and one, since this is the largest
    possible difference between a predicted probability (which must be
    between zero and one) and the actual outcome (which can take on values
    of only 0 and 1). It can be decomposed is the sum of refinement loss and
    calibration loss.

    The Brier score is appropriate for binary and categorical outcomes that
    can be structured as true or false, but is inappropriate for ordinal
```

```
    variables which can take on three or more values (this is because the
    Brier score assumes that all possible outcomes are equivalently
    "distant" from one another). Which label is considered to be the positive
    label is controlled via the parameter `pos_label`, which defaults to
    the greater label unless `y_true` is all 0 or all -1, in which case
    `pos_label` defaults to 1.

    Read more in the :ref:`User Guide <brier_score_loss>`.

    Parameters
    ----------
    y_true : array of shape (n_samples,)
        True targets.

    y_prob : array of shape (n_samples,)
        Probabilities of the positive class.

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

    pos_label : int or str, default=None
        Label of the positive class. `pos_label` will be infered in the
        following manner:

        * if `y_true` in {-1, 1} or {0, 1}, `pos_label` defaults to 1;
        * else if `y_true` contains string, an error will be raised and
          `pos_label` should be explicitely specified;
        * otherwise, `pos_label` defaults to the greater label,
          i.e. `np.unique(y_true)[-1]`.

    Returns
    -------
    score : float
        Brier score loss.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.metrics import brier_score_loss
    >>> y_true = np.array([0, 1, 1, 0])
    >>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
    >>> y_prob = np.array([0.1, 0.9, 0.8, 0.3])
    >>> brier_score_loss(y_true, y_prob)
    0.037...
    >>> brier_score_loss(y_true, 1-y_prob, pos_label=0)
    0.037...
    >>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
    0.037...
    >>> brier_score_loss(y_true, np.array(y_prob) > 0.5)
    0.0

    References
```

```
    ----------
    .. [1] `Wikipedia entry for the Brier score
            <https://en.wikipedia.org/wiki/Brier_score>`_.
    """
    y_true = column_or_1d(y_true)
    y_prob = column_or_1d(y_prob)
    assert_all_finite(y_true)
    assert_all_finite(y_prob)
    check_consistent_length(y_true, y_prob, sample_weight)

    y_type = type_of_target(y_true)
    if y_type != "binary":
        raise ValueError(
            f"Only binary classification is supported. The type of the target "
            f"is {y_type}."
        )

    if y_prob.max() > 1:
        raise ValueError("y_prob contains values greater than 1.")
    if y_prob.min() < 0:
        raise ValueError("y_prob contains values less than 0.")

    try:
        pos_label = _check_pos_label_consistency(pos_label, y_true)
    except ValueError:
        classes = np.unique(y_true)
        if classes.dtype.kind not in ('O', 'U', 'S'):
            # for backward compatibility, if classes are not string then
            # `pos_label` will correspond to the greater label
            pos_label = classes[-1]
        else:
            raise
    y_true = np.array(y_true == pos_label, int)
    return np.average((y_true - y_prob) ** 2, weights=sample_weight)
```