```
from scipy import sparse
import numpy as np
from sklearn import metrics


def micro_f1score(Y_test, Y_pred):
    true_pos_num = Y_pred.multiply(Y_test).sum()
    pos_num = Y_pred.sum()
    true_num = Y_test.sum()

    # Recall
    prec, recall = (
        true_pos_num / np.maximum(pos_num, 1),
        true_pos_num / np.maximum(true_num, 1),
    )
    f1 = 2 * prec * recall / np.maximum(prec + recall, 1e-12)
    return f1


def macro_f1score(Y_test, Y_pred):
    true_pos_num = np.array(Y_pred.multiply(Y_test).sum(axis=0)).reshape(-1)
    pos_num = np.array(Y_pred.sum(axis=0)).reshape(-1)
    true_num = np.array(Y_test.sum(axis=0)).reshape(-1)

    # Recall
    prec, recall = (
        true_pos_num / np.maximum(pos_num, 1),
        true_pos_num / np.maximum(true_num, 1),
    )
    f1 = 2 * prec * recall / np.maximum(prec + recall, 1e-12)
    return np.mean(f1)


def micro_hamming_loss(Y_test, Y_pred):
    intersec = Y_test.multiply(Y_pred)
    errors = Y_test + Y_pred - 2 * intersec
    return errors.mean()


def macro_hamming_loss(Y_test, Y_pred):
    intersec = Y_test.multiply(Y_pred)
    errors = Y_test + Y_pred - 2 * intersec
    return np.mean(errors.mean(axis=0))


def average_precision(Y_test, Y_score):

    Y = Y_test + Y_score
    r, c, _ = sparse.find(Y)
    y_true = np.array(Y_test[(r, c)]).reshape(-1)
    y_score = np.array(Y_score[(r, c)]).reshape(-1)
    n_zeros = np.prod(Y.shape) - len(r)

    # make y_true a boolean vector
    y_true = y_true == 1

    # sort scores and corresponding truth values
    desc_score_indices = np.argsort(y_score, kind="mergesort")[::-1]
    y_score = y_score[desc_score_indices]
```

```python
    y_true = y_true[desc_score_indices]
    weight = 1.0

    # y_score typically has many tied values. Here we extract
    # the indices associated with the distinct values. We also
    # concatenate a value for the end of the curve.
    distinct_value_indices = np.where(np.diff(y_score))[0]
    threshold_idxs = np.r_[distinct_value_indices, y_true.size - 1]

    # accumulate the true positives with decreasing threshold
    tps = np.cumsum(y_true * weight)[threshold_idxs]
    fps = 1 + threshold_idxs - tps

    tps = np.r_[tps, tps[-1]]
    fps = np.r_[fps, fps[-1] + n_zeros]
    thresholds = np.r_[y_score[threshold_idxs], 0]
    precision = tps / (tps + fps)
    precision[np.isnan(precision)] = 0
    recall = tps / tps[-1]

    # stop when full recall attained
    # and reverse the outputs so recall is decreasing
    last_ind = tps.searchsorted(tps[-1])
    sl = slice(last_ind, None, -1)
    precision, recall, thresholds = (
        np.r_[precision[sl], 1],
        np.r_[recall[sl], 0],
        thresholds[sl],
    )
    return -np.sum(np.diff(recall) * np.array(precision)[:-1])


def auc_roc(Y_test, Y_score):

    Y = Y_test + Y_score
    r, c, _ = sparse.find(Y)
    y_true = np.array(Y_test[(r, c)]).reshape(-1)
    y_score = np.array(Y_score[(r, c)]).reshape(-1)
    n_zeros = np.prod(Y.shape) - len(r)

    # make y_true a boolean vector
    y_true = y_true == 1

    # sort scores and corresponding truth values
    desc_score_indices = np.argsort(y_score, kind="mergesort")[::-1]
    y_score = y_score[desc_score_indices]
    y_true = y_true[desc_score_indices]
    weight = 1.0

    # y_score typically has many tied values. Here we extract
    # the indices associated with the distinct values. We also
    # concatenate a value for the end of the curve.
    distinct_value_indices = np.where(np.diff(y_score))[0]
    threshold_idxs = np.r_[distinct_value_indices, y_true.size - 1]

    # accumulate the true positives with decreasing threshold
    tps = np.cumsum(y_true * weight)[threshold_idxs]
    fps = 1 + threshold_idxs - tps
```

```
tps = np.r_[tps, tps[-1]]
fps = np.r_[fps, fps[-1] + n_zeros]
thresholds = np.r_[y_score[threshold_idxs], 0]
# Attempt to drop thresholds corresponding to points in between and
# collinear with other points. These are always suboptimal and do not
# appear on a plotted ROC curve (and thus do not affect the AUC).
# Here np.diff(_, 2) is used as a "second derivative" to tell if there
# is a corner at the point. Both fps and tps must be tested to handle
# thresholds with multiple data points (which are combined in
# _binary_clf_curve). This keeps all cases where the point should be kept,
# but does not drop more complicated cases like fps = [1, 3, 7],
# tps = [1, 2, 4]; there is no harm in keeping too many thresholds.
if len(fps) > 2:
    optimal_idxs = np.where(
        np.r_[True, np.logical_or(np.diff(fps, 2), np.diff(tps, 2)), True]
    )[0]
    fps = fps[optimal_idxs]
    tps = tps[optimal_idxs]
    thresholds = thresholds[optimal_idxs]

# Add an extra threshold position
# to make sure that the curve starts at (0, 0)
tps = np.r_[0, tps]
fps = np.r_[0, fps]
thresholds = np.r_[thresholds[0] + 1, thresholds]

fpr = fps / fps[-1]
tpr = tps / tps[-1]

return metrics.auc(fpr, tpr)
```