```
import numpy as np
from scipy import sparse
from numba import njit
from knn import kNN
from scipy.special import expit


class multilabel_kNN(kNN):
    """Multilabel k-NN
    Zhang, Min-Ling, and Zhi-Hua Zhou. 2007.
    “ML-KNN: A Lazy Learning Approach to Multi-Label Learning.”
    Pattern Recognition 40 (7): 2038–48.
    """

    def __init__(
        self, k=5, metric="euclidean", exact=True, s=1, prior="sample", **params
    ):
        super().__init__(k=k, metric=metric, exact=exact, **params)
        self.s = s
        self.prior_type = prior

    # fit knn model
    def fit(self, X, Y):
        """Fit the model according to the given training data.

        :param X: training data
        :type X: numpy.ndarray or scipy.sparse.csr_matrix
        :param Y: training labels
        :type Y: numpy.ndarray or scipy.sparse.csr_matrix
        :return: self
        :rtype: object
        """
        X, Y = self._homogenize(X, Y)
        self.Y = Y

        self.n_indexed_samples, self.n_labels = Y.shape[0], Y.shape[1]
        _, self.n_features = X.shape[0], X.shape[1]

        #
        # Calculate the prior probabilities
        #
        if self.prior_type == "sample":
            self.priors = (self.s + np.array(Y.sum(axis=0)).reshape(-1)) / (
                self.s * 2 + self.n_indexed_samples
            )
        elif self.prior_type == "uniform":
            self.priors = np.ones(self.n_labels) * 0.5

        #
        # Calculate the posterior probabilities
        #
        # make knn graph
        self._make_faiss_index(X)
        A = self._make_knn_graph(X, self.k + 1, exclude_selfloop=True)

        #
        # Create count vs label table
        #
        self.C_1 = np.zeros((self.k + 1, self.n_labels))
        self.C_0 = np.zeros((self.k + 1, self.n_labels))
        self.C1, self.C_0 = self._count_events(A, Y)

        # Calculate the probabilities
        self.marginal_1 = np.array(np.sum(self.C_1, axis=0)).reshape(-1)
        self.marginal_0 = self.n_indexed_samples - self.marginal_1

        return self

    def predict(self, X, return_prob=False):
        """Predict class labels for samples in X.

        :param X: test data
        :type X: numpy.ndarray or scipy.sparse.csr_matrix
        :return: predicted labels
        :rtype: numpy.ndarray
        """

        X = self._homogenize(X)

        if self.metric == "cosine":
            X = np.einsum("ij,i->ij", X, 1 / np.maximum(np.linalg.norm(X, axis=1), 
1e-32))

        A = self._make_knn_graph(X, self.k, exclude_selfloop=False)

        C = A @ self.Y
        samples, labels, count = sparse.find(C)
        count = count.astype(int)

        # Calculate the posterior probabilities
        safelog = lambda x: np.log(np.maximum(x, 1e-12))
        log_likelihood_1 = safelog(self.priors[labels]) + safelog(
            np.array((self.s + self.C_1[(count, labels)])).reshape(-1)
            / (self.s * (self.k + 1) + self.marginal_1[labels])
        )
        log_likelihood_0 = safelog(1 - self.priors[labels]) + safelog(
            np.array((self.s + self.C_0[(count, labels)])).reshape(-1)
            / (self.s * (self.k + 1) + self.marginal_0[labels])
        )
        pred_positive = log_likelihood_1 > log_likelihood_0
        Ypred = sparse.csr_matrix(
            (pred_positive, (samples, labels)), shape=(X.shape[0], self.n_labels)
        )
        if return_prob:
            prob = expit(-log_likelihood_0 + log_likelihood_1)
            Yprob = sparse.csr_matrix(
                (prob, (samples, labels)), shape=(X.shape[0], self.n_labels)
            )
            return Ypred, Yprob
        else:
            return Ypred

    def _count_events(self, A, Y):
        """
        Calculate the conditional probability p for the binomial distribution.

        params: A: knn graph
        params: Y: label matrix
        return: p1, p0: conditional probability for the binomial distribution
        """
        n_nodes, n_labels = Y.shape[0], Y.shape[1]

        Y.sort_indices()
        C_1, C = _count_neighbors(
            A.indptr, A.indices, Y.indptr, Y.indices, n_nodes, n_labels, self.k
        )
        C_0 = C - C_1
        return C_1, C_0


@njit(nogil=True)
def _isin_sorted(a, x):
    return a[np.searchsorted(a, x)] == x


@njit(nogil=True)
def _neighbors(indptr, indices_or_data, t):
    return indices_or_data[indptr[t] : indptr[t + 1]]


@njit(nogil=True)
def _count_neighbors(
    A_indptr, A_indices, Y_indptr, Y_indices, n_nodes, n_labels, n_neighbors
):
    C1 = np.zeros((n_neighbors + 1, n_labels), dtype=np.int32)
    Call = np.zeros((n_neighbors + 1, n_labels), dtype=np.int32)
    cnt_1 = np.zeros(n_labels, dtype=np.int32)
    cnt_all = np.zeros(n_labels, dtype=np.int32)
    for i in range(n_nodes):
        Y_neighbors_i = _neighbors(Y_indptr, Y_indices, i)
        for j in _neighbors(A_indptr, A_indices, i):  #
            for yj in _neighbors(Y_indptr, Y_indices, j):
                if _isin_sorted(Y_neighbors_i, yj):
                    cnt_1[yj] += 1
                cnt_all[yj] += 1
        for label_id in range(n_labels):
            C1[cnt_1[label_id], label_id] += 1
            Call[cnt_all[label_id], label_id] += 1
        cnt_1 *= 0
        cnt_all *= 0
    return C1, Call
```