```
"""k-nearest neighbor predictor"""
import numpy as np
from scipy import sparse
import faiss


class kNN:
    def __init__(self, k=5, metric="euclidean", exact=True, gpu_id=None):
        self.k = k
        self.metric = metric
        self.gpu_id = gpu_id
        self.exact = exact

    # fit knn model
    def fit(self, X, Y):
        """Fit the model using X as training data and Y as target values

        :param X: training data
        :type X: numpy.ndarray
        :param Y: target values
        :type Y: numpy.ndarray
        :return: self
        :rtype: object
        """
        # make knn graph
        X = self._homogenize(X)
        self.n_indexed_samples = X.shape[0]
        self._make_faiss_index(X)
        self.labels = None

        if isinstance(Y, np.ndarray):
            self.labels, label_ids = np.unique(Y, return_inverse=True)
            n_labels = len(self.labels)
            Y = sparse.csr_matrix(
                (np.ones_like(Y), (np.arange(len(Y)), label_ids)),
                shape=(len(Y), n_labels),
            )
        self.Y = Y
        return self

    def predict(self, X, return_prob=False, return_ids=False):
        """Predict the class labels for the provided data

        :param X: data to predict
        :type X: numpy.ndarray
        :return: predicted class labels
        :rtype: numpy.ndarray
        """

        X = self._homogenize(X)

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

        C = A @ self.Y
        cids = np.array(np.argmax(C, axis=1)).reshape(-1)

        if return_ids:
            C.data /= self.k
            C = np.array(C[(np.arange(C.shape[0]), cids)]).reshape(-1)
```

```
            if self.labels is not None:
                Ypred = self.labels[cids]
            else:
                Ypred = cids

        else:
            Ypred = sparse.csr_matrix(
                (np.ones_like(cids), (np.arange(len(cids)), cids)),
                shape=(len(cids), self.Y.shape[1]),
            )
            C.data /= self.k

        if return_prob:
            return Ypred, C
        else:
            return Ypred

    def _make_faiss_index(self, X):
        """Create an index for the provided data

        :param X: data to index
        :type X: numpy.ndarray
        :raises NotImplementedError: if the metric is not implemented
        :return: faiss index
        :rtype: faiss.Index
        """
        n_samples, n_features = X.shape[0], X.shape[1]
        X = X.astype("float32")
        if n_samples < 1000:
            self.exact = True

        index = (
            faiss.IndexFlatL2(n_features)
            if self.metric == "euclidean"
            else faiss.IndexFlatIP(n_features)
        )

        if not self.exact:
            # code_size = 32
            train_sample_num = np.minimum(100000, X.shape[0])
            nlist = int(np.ceil(np.sqrt(train_sample_num)))
            faiss_metric = (
                faiss.METRIC_L2
                if self.metric == "euclidean"
                else faiss.METRIC_INNER_PRODUCT
            )
            # index = faiss.IndexIVFPQ(
            #    index, n_features, nlist, code_size, 8, faiss_metric
            # )
            index = faiss.IndexIVFFlat(index, n_features, nlist, faiss_metric)
            # index.nprobe = 5

        if self.gpu_id is not None:
            res = faiss.StandardGpuResources()
            index = faiss.index_cpu_to_gpu(res, self.gpu_id, index)

        if not index.is_trained:
            Xtrain = X[
```

```
                np.random.choice(X.shape[0], train_sample_num, replace=False), :
            ].copy(order="C")
            index.train(Xtrain)

        index.add(X)
        self.index = index

    def _make_knn_graph(self, X, k, exclude_selfloop=True, weighted=False):
        """Construct the k-nearest neighbor graph

        :param X: data to construct the graph
        :type X: numpy.ndarray
        :param k: number of neighbors
        :type k: int
        :param exclude_selfloop: whether to exclude self-loops, defaults to True
        :type exclude_selfloop: bool, optional
        :return: k-nearest neighbor graph
        :rtype: numpy.ndarray
        """
        # get the number of samples and features
        n_samples, n_features = X.shape

        # create a list of k nearest neighbors for each vector
        dist, indices = self.index.search(X.astype("float32"), k)

        rows = np.arange(n_samples).reshape((-1, 1)) @ np.ones((1, k))

        # create the knn graph
        rows, indices, dist = rows.ravel(), indices.ravel(), dist.ravel()
        if exclude_selfloop:
            s = rows != indices
            rows, indices, dist = rows[s], indices[s], dist[s]

        s = indices >= 0
        rows, indices, dist = rows[s], indices[s], dist[s]

        if weighted is False:
            dist = dist * 0 + 1

        A = sparse.csr_matrix(
            (dist, (rows, indices)),
            shape=(n_samples, self.n_indexed_samples),
        )
        return A

    def _homogenize(self, X, Y=None):
        if self.metric == "cosine":
            X = np.einsum("ij,i->ij", X, 1 / np.maximum(np.linalg.norm(X, axis=1),
1e-32))
        X = X.astype("float32")

        if X.flags["C_CONTIGUOUS"]:
            X = X.copy(order="C")

        if Y is not None:
            if sparse.issparse(Y):
                if not sparse.isspmatrix_csr(Y):
                    Y = sparse.csr_matrix(Y)
            elif isinstance(Y, np.ndarray):
```

```
                Y = sparse.csr_matrix(Y)
            else:
                raise ValueError("Y must be a scipy sparse matrix or a numpy
array")
            Y.data[Y.data != 1] = 1
            return X, Y
        else:
            return X
```