```
import operator

import numpy as np
from past.builtins import xrange


class KNearestNeighbor(object):
    """ a kNN classifier with L2 distance """

    def __init__(self):
        pass

    def train(self, X, y):
        """
    Train the classifier. For k-nearest neighbors this is just
    memorizing the training data.

    Inputs:
    - X: A numpy array of shape (num_train, D) containing the training data
      consisting of num_train samples each of dimension D.
    - y: A numpy array of shape (N,) containing the training labels, where
         y[i] is the label for X[i].
    """
        self.X_train = X  # m here (5000, 3072)
        self.y_train = y  # m here (500,)

    def predict(self, X, k=1, num_loops=0):
        """
    Predict labels for test data using this classifier.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data consisting
         of num_test samples each of dimension D.
    - k: The number of nearest neighbors that vote for the predicted labels.
    - num_loops: Determines which implementation to use to compute distances
      between training points and testing points.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
        if num_loops == 0:
            dists = self.compute_distances_no_loops(X)
        elif num_loops == 1:
            dists = self.compute_distances_one_loop(X)
        elif num_loops == 2:
            dists = self.compute_distances_two_loops(X)
        else:
            raise ValueError('Invalid value %d for num_loops' % num_loops)

        return self.predict_labels(dists, k=k)

    def compute_distances_two_loops(self, X):
        """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    # m here (500, 3072)

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    # m here (500, 5000)
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
        num_test = X.shape[0]  # m here num_test=500
```

```
        num_train = self.X_train.shape[0]  # m here num_train=5000
        dists = np.zeros((num_test, num_train))
        for i in xrange(num_test):  # m 0 to 499
            for j in xrange(num_train):  # m 0 to 4999
                #####################################################################
                # m completed                                                           #
                # Compute the l2 distance between the ith test point and the jth    #
                # training point, and store the result in dists[i, j]. You should   #
                # not use a loop over dimension.                                    #
                #####################################################################
                test_image = X[i]
                memorized_training_image = self.X_train[j]
                dists[i, j] = np.sqrt(np.sum(np.square(test_image - memorized_training_image)))
                #####################################################################
                #                       END OF YOUR CODE                            #
                #####################################################################
        return dists

    def compute_distances_one_loop(self, X):
        """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in xrange(num_test):
            #######################################################################
            # m done but takes  longer than two loops:
#
            # Compute the l2 distance between the ith test point and all training #
            # points, and store the result in dists[i, :].                        #
            #######################################################################
            test_image = X[i]
            all_memorized_training_images = self.X_train
            dists[i, :] = np.sqrt(np.sum(np.square(all_memorized_training_images - test_image),
axis=1))
            #######################################################################
            #                         END OF YOUR CODE                            #
            #######################################################################
        return dists

    def compute_distances_no_loops(self, X):
        """
    Compute the distance between each test point in X and each training point
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        #########################################################################
        # m done:                                                                 #
        # Compute the l2 distance between all test points and all training      #
        # points without using any explicit loops, and store the result in      #
        # dists.                                                                #
        #                                                                       #
        # You should implement this function using only basic array operations; #
        # in particular you should not use functions from scipy.                #
        #                                                                       #
        # HINT: Try to formulate the l2 distance using matrix multiplication    #
        #       and two broadcast sums.                                         #
        #########################################################################
        # dists = np.sqrt(np.sum(X**2, axis=1).reshape(num_test, 1) + np.sum(self.X_train**2,
axis=1) - 2 * X.dot(self.X_train.T))
        a_squared = np.sum(X ** 2, axis=1).reshape(num_test, 1)
```

```
        b_squared = np.sum(self.X_train ** 2, axis=1)
        two_a_b = 2 * X.dot(self.X_train.T)
        dists = np.sqrt(a_squared + b_squared - two_a_b)
        #########################################################################
        #                         END OF YOUR CODE                              #
        #########################################################################
        return dists

    def predict_labels(self, dists, k=1):
        """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
        # m here dists = (500, 5000)
        # m here return is (500,)
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in xrange(num_test):
            # A list of length k storing the labels of the k nearest neighbors to
            # the ith test point.
            closest_y = []
            #########################################################################
            # m done                                                                 #
            # Use the distance matrix to find the k nearest neighbors of the ith    #
            # testing point, and use self.y_train to find the labels of these       #
            # neighbors. Store these labels in closest_y.                           #
            # Hint: Look up the function numpy.argsort.                             #
            #########################################################################
            sorted_indices = np.argsort(dists[i])
            selected_closest_neighbours = list(sorted_indices[0:k])
            for neighbour in selected_closest_neighbours:
                closest_y.append(self.y_train[neighbour])
            #########################################################################
            # m done:                                                                 #
            # Now that you have found the labels of the k nearest neighbors, you    #
            # need to find the most common label in the list closest_y of labels.   #
            # Store this label in y_pred[i]. Break ties by choosing the smaller     #
            # label.                                                                #
            #########################################################################
            y_pred[i] = self.mostCommonLabel(closest_y)
            #########################################################################
            #                           END OF YOUR CODE                            #
            #########################################################################

        return y_pred

    @staticmethod
    def mostCommonLabel(items):
        counts = {}
        counted = []
        for item in items:
            if item in counted:
                continue
            counted.append(item)
            counts[item] = (items.count(item))

        sorted_items = sorted(counts.items(), key=operator.itemgetter(1))
        sorted_items.reverse()
        first_item = sorted_items[0][0]
        highest = first_item
        try:
```

```
        second_item = sorted_items[1][0]
        if sorted_items[0][1] == sorted_items[1][1]:
            length_of_first_item = len(first_item)
            length_of_second_item = len(second_item)

            if length_of_first_item > length_of_second_item:
                print("first item is", first_item, "length is", length_of_first_item)
                print("second item is", second_item, "length is", length_of_second_item)
                print(first_item, "length is greater than", second_item, "length")
                highest = second_item
                print("highest is ", highest)
        return highest
    except:
        return highest
```