```
import numpy as np
import pdb


class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    - norm: the function with which the norm is taken.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))
      #norm = 2

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    # print("X.shape:", X.shape)
    # print("X_train.shape:", self.X_train.shape)

    for i in np.arange(num_test):

      for j in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Compute the distance between the ith test point and the jth
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #

        dists[i, j] = norm(X[i] - self.X_train[j])

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

    return dists

  # def compute_Ln_distances_vectorized(self, X, norm=None):
  #   if norm is None:
  #     norm = lambda x: np.sqrt(np.sum(x**2))
  #     #norm = 2
```

```
  #   num_test = X.shape[0]
  #   num_train = self.X_train.shape[0]

  #   dists = norm(X[:,np.newaxis,:] - self.X_train)

  #   return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #   NOT use a for loop (or list comprehension).  You may only use
    #   numpy operations.
    #
    #   HINT: use broadcasting.  If you have a shape (N,1) array and
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ================================================================ #

    dists = np.sqrt(np.sum(np.square(X)[:,np.newaxis,:], axis=2) - 2 * X@self.X_train.T + np.s
um(np.square(self.X_train), axis=1))

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    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].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in np.arange(num_test):
```

```
        # A list of length k storing the labels of the k nearest neighbors to
        # the ith test point.
        closest_y = []
        # ================================================================ #
        # YOUR CODE HERE:
        #   Use the distances to calculate and then store the labels of
        #   the k-nearest neighbors to the ith test point.  The function
        #   numpy.argsort may be useful.
        #
        #   After doing this, find the most common label of the k-nearest
        #   neighbors.  Store the predicted label of the ith training example
        #   as y_pred[i].  Break ties by choosing the smaller label.
        # ================================================================ #

        closest_y = self.y_train[np.argpartition(dists[i,:], k)[:k]]
        vals, counts = np.unique(closest_y, return_counts=True)
        y_pred[i] = vals[np.argmax(counts)]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

      return y_pred
```