```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and modified for ECE C147/
C247 at UCLA.
"""

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))
    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(self.X_train[j] - X[i])

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

    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.
        # ================================================================ #

    # Let A = X, B = self.X_train
    # (A-B)**2 = A**2 - 2AB + B**2

    # Have: A = (500, 3072), B = (5000, 3072)
    # Want: A**2 = (500, 1), B**2 = (5000,), AB = (500, 5000)

    # (500,) array with each row being the sum of a testing data point's coordinates
    a_squared = np.sum(np.square(X), axis=1)
    # Reshape to (500,1) array
    a_squared = a_squared.reshape((a_squared.shape[0], 1))
    # (5000,) array with each row being the sum of a test data point's coordinates squared
    b_squared = np.sum(np.square(self.X_train), axis=1)
    # (500,1) array + (5000,) array = (500, 5000) array that holds A**2 + B**2 for each
    # pair of testing points and training points
    dists = a_squared + b_squared

    # (500, 3072).T * (5000, 3072) = (500, 3072) * (3072, 5000) = (500, 5000)
    a_times_b = np.dot(X, self.X_train.T)
    dists -= 2*a_times_b
    dists = np.sqrt(dists)

        # ================================================================ #
        # 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_idxs = np.argsort(dists[i])[:k]
    closest_y = [self.y_train[idx] for idx in closest_idxs]
    unique, counts = np.unique(closest_y, return_counts=True)
    closest_y = sorted(dict(zip(unique, counts)).items(), key=lambda x: (-x[1], x[0]))
    y_pred[i] = closest_y[0][0]

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

  return y_pred
```