```python
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))
    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].
        # ================================================================ #

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

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

```python
        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.
        # ================================================================ #

        trainsquared = np.sum(self.X_train**2, axis=1, keepdims=True)
        testsquared = np.sum(X**2, axis=1)
        multiplied = np.dot(self.X_train, X.T)
        dists = np.sqrt(trainsquared - 2*multiplied + testsquared)
        dists = dists.T

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

        return dists

    def compute_L1_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]
```

```python
    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.
    # ================================================================ #
    for i in range(0,num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.X_train - X[i,:]), axis = 1)
      dists[i]=distances


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

    return dists

  def compute_Linf_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.
    # ================================================================ #
    for i in range(0,num_test):
      # find the nearest training image to the i'th test image
```

```
        # using the L1 distance (sum of absolute value differences)
        distances = np.sum(np.linalg.norm(self.X_train - X[i,:],ord="inf"), axis = 1)
        dists[i]=distances


      # ================================================================ #
      # 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.
        # ================================================================ #
        distance = dists[i]
        sortIndices = np.argsort(distance)
        bestOnes = sortIndices[0:k]
        closest_y = self.y_train[bestOnes]
        counts = np.bincount(closest_y)
        y_pred[i]=np.argmax(counts)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

      return y_pred
```