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

self.y\_train = y

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.

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))

for i in range(num\_test):

for j in range(num\_train):

#####################################################################

# TODO: #

# 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. #

#####################################################################

dists[i,j]=np.sqrt(np.sum(X[i,:]-self.X\_train[j,:]\*\*2))

#####################################################################

# 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 range(num\_test):

#######################################################################

# TODO: #

# Compute the l2 distance between the ith test point and all training #

# points, and store the result in dists[i, :]. #

#######################################################################

dists[i,:]=np.sqrt(np.sum((X[i,:]-self.X\_train)\*\*2,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))

#########################################################################

# TODO: #

# 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. #

#########################################################################

X\_squared=np.sum(X\*\*2,axis=1)

Y\_squared=np.sum(self.X\_ratin\*\*2,axis=1)

XY=np.dot(X,self.X\_train.T)

dists=np.sqrt(X\_squared[:,np.newaxis]+Y\_squared-2\*XY)

#########################################################################

# 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].

"""

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 = []

#########################################################################

# TODO: #

# 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. #

#########################################################################

closest\_y = self.y\_train[np.argsort(dists[i,:])[0:k]]

#########################################################################

# TODO: #

# 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] = np.argmax(np.bincount(closest\_y))

#########################################################################

# END OF YOUR CODE #

#########################################################################

return y\_pred