RepMLA_Etivity1_ALISONOCONNOR_0427845

July 25, 2021

#**Artificial Intelligence - MSc** CS6501 - MACHINE LEARNING AND APPLICATIONS #**Business Analytics - MSc** ET5003 - MACHINE LEARNING APPLICATIONS ##***Annual Repeat*** ###Instructor: Enrique Naredo

###RepMLA_Lab-1.13

Student ID: 0427845

Student name: Alison O'Connor

1 E-tivity: K-Nearest Neighbors

1.1 Overview

The goal is to implement the K-nearest neighbors algorithm (a supervised machine learning algorithm) and apply it to a real dataset. Along the way you should familiarize yourself with some of the terminology you have read in the note. You will also get a chance to practice with Python and working with large datasets.

1.2 Dataset introduction

The handwritten digit recognition is the ability of computers to recognize human handwritten digits. It is a hard task for the machine because handwritten digits are not perfect and can be made with many different flavors. The handwritten digit recognition is the solution to this problem which uses the image of a digit and recognizes the digit present in the image.

The dataset for this E-tivity is a set of handwritten digits from zip codes written on hand-addressed letters (MNIST-Modified National Institute of Standards and Technology database).

Read about this dataset by going to the Elements of Statistical Learning website, ESL, then clicking on the `Data` tab, then clicking on the `Info` for the zip code dataset (the last dataset).

Use the command less in the terminal to view the beginning of each file. Both datasets have the same format: the first column is the "label" (or class) (here an integer between 0 and 9, inclusive, that corresponds to the identity of a hand-written zip code digit), and the rest of each row is made up of gray-scale values corresponding to the image of this hand-written digit.

One useful technique is to load a dataset from a file into a numpy array. Here is an example:

[9]:
```
import numpy as np
import os
import sys
import matplotlib.pyplot as plt
from sklearn import cluster
```

[3]:
```
##CHECK CURRENT WORK DIRECTORY.
cwd=os.getcwd()

train_data = np.loadtxt("mnist.train") #"path/to/train/file"
test_data  = np.loadtxt("mnist.test")

print(train_data.shape)
print(test_data.shape)
```

```
(7291, 257)
(2007, 257)
```

[ ]:
```
train_data #each row is a different image
```

[ ]:
```
array([[ 6.   , -1.   , -1.   , …, -1.   , -1.   , -1.   ],
       [ 5.   , -1.   , -1.   , …, -0.671, -0.828, -1.   ],
       [ 4.   , -1.   , -1.   , …, -1.   , -1.   , -1.   ],
       …,
       [ 3.   , -1.   , -1.   , …, -1.   , -1.   , -1.   ],
       [ 0.   , -1.   , -1.   , …, -1.   , -1.   , -1.   ],
       [ 1.   , -1.   , -1.   , …, -1.   , -1.   , -1.   ]])
```

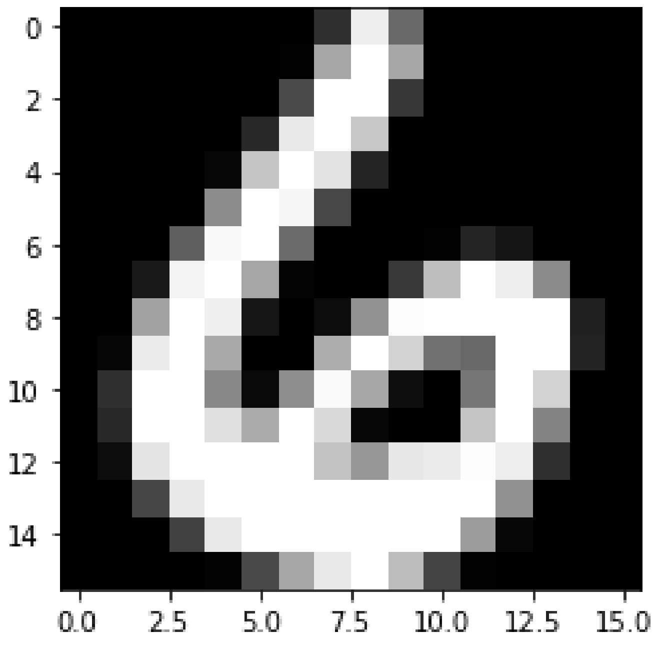
The first column is the class, it tells us which digit we will find in the image, as you can see hereafter for the first row (it is a 6):

[ ]:
```
import matplotlib.pyplot as plt
row=0
flatten_image=train_data[row,1:]# each 16x16 image has been flatten into a␣
 ↪vector
im=flatten_image.reshape(16,16)
print(im)#as a matrix
plt.gray()
plt.imshow(im)#as an image
```

```
[[-1.    -1.    -1.    -1.    -1.    -1.    -1.    -0.631  0.862 -0.167
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.    -1.    -1.    -0.992  0.297  1.     0.307
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.    -1.    -1.    -0.41   1.     0.986 -0.565
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.    -1.    -0.683  0.825  1.     0.562 -1.
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.    -0.938  0.54   1.     0.778 -0.715 -1.
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.     0.1    1.     0.922 -0.439 -1.    -1.
```

```
  -1.    -1.    -1.    -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -0.257  0.95   1.    -0.162 -1.    -1.    -1.
  -0.987 -0.714 -0.832 -1.    -1.    -1.   ]
 [-1.    -1.    -0.797  0.909  1.     0.3   -0.961 -1.    -1.    -0.55
   0.485  0.996  0.867  0.092 -1.    -1.   ]
 [-1.    -1.     0.278  1.     0.877 -0.824 -1.    -0.905  0.145  0.977
   1.     1.     1.     0.99  -0.745 -1.   ]
 [-1.    -0.95   0.847  1.     0.327 -1.    -1.     0.355  1.     0.655
  -0.109 -0.185  1.     0.988 -0.723 -1.   ]
 [-1.    -0.63   1.     1.     0.068 -0.925  0.113  0.96   0.308 -0.884
  -1.    -0.075  1.     0.641 -0.995 -1.   ]
 [-1.    -0.677  1.     1.     0.753  0.341  1.     0.707 -0.942 -1.
  -1.     0.545  1.     0.027 -1.    -1.   ]
 [-1.    -0.903  0.792  1.     1.     1.     1.     0.536  0.184  0.812
   0.837  0.978  0.864 -0.63  -1.    -1.   ]
 [-1.    -1.    -0.452  0.828  1.     1.     1.     1.     1.     1.
   1.     1.     0.135 -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -0.483  0.813  1.     1.     1.     1.     1.
   1.     0.219 -0.943 -1.    -1.    -1.   ]
 [-1.    -1.    -1.    -1.    -0.974 -0.429  0.304  0.823  1.     0.482
  -0.474 -0.991 -1.    -1.    -1.    -1.   ]]
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f4e93a293c8>
```



The test set is similar:

[ ]:
```
test_data
```

```
[ ]: array([[ 9., -1., -1., …, -1., -1., -1.],
            [ 6., -1., -1., …, -1., -1., -1.],
            [ 3., -1., -1., …, -1., -1., -1.],
            …,
            [ 4., -1., -1., …, -1., -1., -1.],
            [ 0., -1., -1., …, -1., -1., -1.],
            [ 1., -1., -1., …, -1., -1., -1.]])
```

1.3 Step 2: Filter the Data

To start, we will just consider two classes, but here we have 10. We will get to such problems later, but for now, devise a way to retain only the rows which have label 2 or 3. Do this for both the train and test data.

One important note: it may be convenient to relabel the 2 to -1 and the 3 to +1, since this will work better with our methods later on (but you do not have to do this).

```
[20]: #FILTER THE TRAINING DATA
sub_train=train_data[(train_data[:,0]<=3) & (train_data[:,0]>=2)]
# print(sub_train[100:150])
##REPLACE ALL VALUES IN FIRST COLUMN 3=1 AND 2=-1
##Run np where on first column of sub_train (returns 1D array same num of items␣
↪as rows of sub_train)
a=np.where(sub_train[:,0]==3, 1, -1)
##replace first column of subtrain with new array
sub_train[:,0]=a
print(sub_train[100:120])
```

```
[[ 1.    -1.    -1.    … -1.    -1.    -1.   ]
 [-1.    -1.    -1.    … -0.983 -1.    -1.   ]
 [-1.    -1.    -1.    …  1.    -0.445 -1.   ]
 …
 [ 1.    -1.    -1.    …  0.239 -0.499 -1.   ]
 [-1.    -1.    -1.    … -0.804  0.189  0.068]
 [-1.    -1.    -1.    … -1.    -1.    -1.   ]]
```

```
[5]: #FILTER THE TEST DATA
sub_test=test_data[(test_data[:,0]<=3) & (test_data[:,0]>=2)]
# print(sub_test[200:250])
a=np.where(sub_test[:,0]==3, 1, -1) #CREATE THE REPLACEMENT ARRAY
sub_test[:,0]=a
# print(sub_test[200:250])
```

1.4 Implement K-nearest neighbors

The main goal of the E-tivity is to implement the K-nearest neighbors classifier to predict the class of each example from the test dataset. Exactly how you implement this part is up to you, but your code should be decomposed into functions, well commented, and easy to understand. Here are some suggestions:

Classification Create a function that takes as input the train set, (at least) a test example and an integer K, and outputs a prediction based on a nearest-neighbor classifier. This function will loop through all the training examples, find the distance between each one and the input test example, and then find the K nearest neighbors (you are welcome to use numpy sorting methods, but look up how they work first). For this subroutine, we will need a distance function. In practice, you have to implement the algorithm 3 in Duame (pag. 33).

You should implement a Python function like:

[30]:
```
def KNN(train,test,K):
    ##USE SKLEARN TO RUN KMEANS FOR A NUMBER OF K CLUSTERS
    clustered_data=cluster.KMeans(n_clusters=K, n_init=10, max_iter=300).
 ↪fit(test)

    return clustered_data.labels_
```

<font color=red < I'm confused. The first item in each row of our arrays is the known class of the image. So if we run KMeans on the data as it currently is then the Euclidean distance will always find the 'nearest neighbour' to be the correct class because the class is in the array. Should we only be using this part of the array [:, 1:end]? /font>

[31]:
```
##FOR A NUMBER OF K CLUSTERS
##EVERY ROW OF THE TEST SET SHOULD BE PUT THROUGH THE ALGORITHM
##

k_list=[1, 2, 5, 10]
labels={}

##FOR SET NUMBER OF CLUSTERS
for K in k_list:
    ##APPEND THE NUM CLUSTERS AND LABELS TO LABELS DICTIONARY
    labels[K]=KNN(sub_train, sub_test, K)
```

```
C:\Users\alison\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:881:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=2.
  warnings.warn(
```

[32]:
```
##I HAVE NO IDEA WHAT THESE ARRAYS MEAN.

labels
```

[32]:
```
{1: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 2: array([0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0,
        0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1,
        1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0,
        0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0,
        0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1,
        1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
        1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0,
        0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1,
        1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1,
        0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0,
        0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
        1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1,
        1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1,
        0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
        1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
        0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0]),
 5: array([0, 3, 0, 2, 3, 4, 3, 0, 0, 4, 0, 4, 3, 1, 3, 0, 4, 1, 1, 3, 3, 4,
        2, 1, 0, 0, 3, 0, 0, 0, 3, 3, 0, 3, 2, 0, 3, 0, 2, 4, 0, 1, 2, 3,
        3, 4, 1, 0, 3, 2, 2, 0, 3, 3, 0, 4, 2, 1, 1, 2, 2, 2, 3, 3, 4, 2,
        2, 0, 2, 4, 3, 1, 4, 0, 2, 1, 4, 4, 4, 2, 3, 2, 0, 2, 3, 4, 0, 2,
        0, 2, 2, 1, 4, 0, 4, 2, 3, 0, 1, 3, 2, 3, 2, 4, 0, 4, 3, 0, 1, 3,
        1, 1, 0, 1, 2, 3, 4, 1, 1, 4, 0, 3, 0, 1, 3, 3, 0, 2, 2, 0, 0, 4,
        1, 4, 1, 3, 0, 0, 2, 2, 2, 2, 0, 0, 1, 2, 2, 3, 0, 2, 1, 4, 2, 0,
        0, 1, 2, 1, 3, 1, 1, 1, 0, 1, 4, 0, 2, 0, 2, 1, 1, 0, 3, 0, 3, 1,
        1, 4, 4, 1, 0, 0, 2, 3, 0, 4, 1, 0, 2, 2, 2, 2, 2, 2, 2, 0, 1, 2,
        2, 2, 2, 0, 2, 2, 2, 4, 4, 1, 0, 4, 1, 3, 3, 4, 0, 2, 4, 4, 2, 1,
        0, 1, 4, 0, 2, 2, 3, 2, 4, 4, 2, 0, 2, 1, 4, 1, 2, 1, 1, 1, 1, 0,
        2, 3, 2, 3, 1, 0, 1, 0, 1, 3, 2, 2, 0, 1, 0, 0, 3, 2, 2, 0, 0, 2,
        4, 2, 2, 2, 2, 0, 1, 2, 0, 2, 2, 4, 1, 3, 0, 2, 1, 4, 2, 0, 1, 4,
        1, 2, 2, 0, 0, 4, 4, 3, 3, 3, 3, 3, 1, 3, 2, 2, 1, 1, 4, 4, 4, 4,
        2, 4, 4, 0, 4, 3, 3, 0, 3, 3, 0, 0, 0, 4, 1, 1, 0, 2, 2, 2, 0, 2,
        3, 4, 1, 2, 1, 1, 0, 4, 4, 3, 4, 4, 2, 0, 2, 0, 4, 4, 4, 0, 3, 1,
      2, 2, 3, 1, 1, 2, 2, 0, 4, 0, 3, 2]),
 10: array([5, 2, 0, 6, 4, 4, 4, 7, 0, 3, 8, 2, 2, 2, 4, 0, 2, 1, 1, 4, 4, 3,
      6, 2, 7, 0, 2, 8, 8, 8, 4, 4, 3, 4, 5, 5, 4, 5, 5, 3, 8, 4, 6, 4,
      4, 3, 1, 7, 2, 6, 9, 7, 4, 2, 7, 2, 5, 1, 1, 6, 3, 5, 4, 4, 4, 9,
      0, 0, 6, 3, 4, 2, 2, 8, 6, 1, 3, 3, 3, 0, 2, 9, 0, 9, 2, 2, 0, 5,
      0, 6, 0, 1, 2, 7, 2, 5, 4, 9, 1, 2, 6, 2, 5, 3, 7, 3, 2, 5, 2, 4,
      1, 1, 0, 3, 6, 4, 3, 1, 2, 2, 8, 4, 8, 1, 4, 4, 0, 9, 9, 5, 8, 3,
      2, 2, 2, 4, 0, 8, 5, 5, 9, 6, 8, 7, 2, 6, 6, 6, 8, 9, 1, 2, 9, 8,
      7, 2, 5, 1, 4, 1, 1, 2, 8, 1, 3, 7, 5, 5, 5, 2, 1, 8, 4, 8, 4, 1,
      2, 3, 3, 1, 7, 7, 9, 4, 8, 3, 2, 6, 5, 5, 9, 6, 6, 0, 9, 7, 1, 9,
      9, 9, 9, 7, 5, 6, 6, 4, 2, 1, 5, 3, 1, 4, 2, 3, 0, 5, 2, 2, 5, 2,
      0, 1, 2, 8, 6, 9, 4, 5, 3, 3, 6, 0, 5, 1, 2, 1, 9, 1, 2, 1, 1, 7,
      9, 2, 9, 4, 1, 8, 1, 0, 1, 4, 9, 9, 0, 1, 7, 7, 4, 5, 9, 0, 8, 9,
      3, 6, 5, 6, 5, 0, 1, 9, 9, 5, 5, 3, 1, 4, 5, 5, 1, 3, 6, 7, 1, 2,
      1, 6, 6, 0, 7, 4, 3, 4, 4, 4, 4, 4, 1, 6, 6, 6, 1, 1, 2, 3, 2, 3,
      6, 2, 2, 7, 2, 4, 4, 7, 4, 4, 9, 8, 7, 2, 1, 1, 8, 6, 6, 6, 7, 6,
      2, 3, 1, 5, 1, 1, 7, 3, 3, 4, 3, 3, 5, 0, 5, 7, 2, 2, 2, 7, 4, 1,
      6, 6, 4, 1, 1, 6, 6, 8, 2, 7, 4, 5])}
```

Distance function An important part of many machine learning methods is the concept of "distance" between examples. We often phrase this as a "metric" on our inputs. Create a function that takes as input two examples (any two examples, although in this case we will use it with one test and one train), and outputs the distance (we will use Euclidean for now) between them. Although there are many built-in functions the perform this task, please implement your distance function from scratch. However, you are welcome to use numpy functions as part of it (for example, you may use np.sum and similar functions, but look up how they work first).

Quantify the accuracy Loop through all the filtered test examples, using your classification function to predict the label for each one. Also create a way of determining if the prediction was correct or not, based on the labels of the test data. Compute the fraction or percentage of correctly predicted examples. How does this change as K varies? Try K 1-10 (at least) and record the accuracy.

1.5 Questions:

- What is the accuracy of KNN in the test set for K=1?
- What is the accuracy of KNN in the test set for K=2?
- What is the accuracy of KNN in the test set for K=3?
- ...
- What is the accuracy of KNN in the test set for K=10?

1.6 In depth questions

- Extend your algorithm to a multi-class setting (i.e. distinguish between 3 or more digits). How does this change your best value of K?
- If you are familiar with confusion matrices, create one for this test dataset and your "best" value of K.
- Create a plot of accuracy vs. K.

- Visualize some of the examples that were classified incorrectly. The examples are 16x16 gray-scale images, so you can plot them on a grid.

Analysis Questions

- What values of k did you try?
- Which value of k produced the highest accuracy? What general trends did you observe as k increased?
- When using the entire training dataset, what are your observations about the runtime of K-nearest neighbors? List 1-2 ideas for making this algorithm faster.