RepMLA_Etivity1_ALISONOCONNOR_0427845_V1

July 29, 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 AOC NOTES:

This is an updated version of the original file submitted after the deadline was extended.

Personal notes by me in markdown cells are in red font to differentiate my notes from Naredo's

2 E-tivity: K-Nearest Neighbors

2.1 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.

2.2 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:

3 Import statements

Use numpy and pandas for data manipulation Use os and sys to access files located in my directory (i.e. accessing working tree) Use matplotlib for plotting

[1]:
```
import numpy as np
import pandas as pd
import os
import sys
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score #CALCULATE ACCURACY
from sklearn.metrics import confusion_matrix #CALCULATE CONFUSION
import time
```

[2]:
```
##CHECK CURRENT WORK DIRECTORY.
cwd=os.getcwd()
print(cwd)

##LOAD DATA INTO NUMPY ARRAY
train_data = np.loadtxt("mnist.train")
test_data  = np.loadtxt("mnist.test")

##CHECK SIZE OF THE ARRAYS
print(test_data.shape, test_data.shape)

##FOR DF WE NEED COL HEADINGS USE INT VALUES TO GIVE DATA HEADINGS
##FIRST COL IS ACTUALLY THE CLASSIFICATION SO WE'LL GIVE THAT A NAME NOT A␣
 ↪NUMBER
col_head=[i for i in range(1, train_data.shape[1], 1)]
##ADD IN CLASS HEADING FOR THE FIRST COLUMM
col_head.insert(0, 'CLASS')

##CONVERT TO DF
train_df=pd.DataFrame(train_data, columns=col_head)
test_df=pd.DataFrame(test_data, columns=col_head)

##ENSURE CLASS COLUMN IS INTEGER TYPE DATA
train_df['CLASS']=train_df['CLASS'].astype(int)
test_df['CLASS']=test_df['CLASS'].astype(int)


##SHOW FIRST FIVE ROWS OF TRAINING SET
train_df.head(5)
```

```
C:\Users\alison\PycharmProjects\ML_APP_UL\WEEK_1\Etivity-1
(2007, 257) (2007, 257)
```

[2]:
```
   CLASS    1    2    3      4      5      6      7      8      9  …    247  \
0      6 -1.0 -1.0 -1.0 -1.000 -1.000 -1.000 -1.000 -0.631  0.862  …  0.304
1      5 -1.0 -1.0 -1.0 -0.813 -0.671 -0.809 -0.887 -0.671 -0.853  … -0.671
2      4 -1.0 -1.0 -1.0 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000  … -1.000
3      7 -1.0 -1.0 -1.0 -1.000 -1.000 -0.273  0.684  0.960  0.450  … -0.318
4      3 -1.0 -1.0 -1.0 -1.000 -1.000 -0.928 -0.204  0.751  0.466  …  0.466

     248    249    250    251    252    253    254    255  256
0  0.823  1.000  0.482 -0.474 -0.991 -1.000 -1.000 -1.000 -1.0
1 -0.671 -0.033  0.761  0.762  0.126 -0.095 -0.671 -0.828 -1.0
2 -1.000 -1.000 -0.109  1.000 -0.179 -1.000 -1.000 -1.000 -1.0
3  1.000  0.536 -0.987 -1.000 -1.000 -1.000 -1.000 -1.000 -1.0
4  0.639  1.000  1.000  0.791  0.439 -0.199 -0.883 -1.000 -1.0

[5 rows x 257 columns]
```

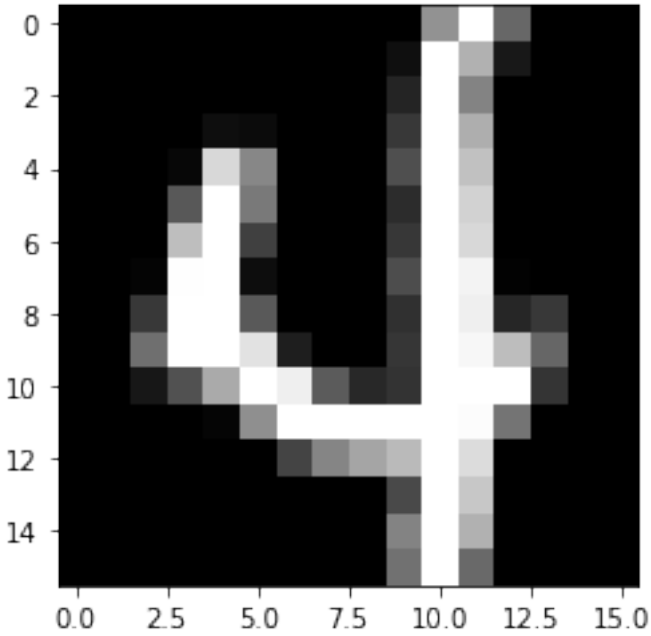
[3]:
```
test_df.head()
```

[3]:
```
   CLASS    1    2    3      4    5      6      7      8      9  …    247  \
0      9 -1.0 -1.0 -1.0 -1.000 -1.0 -0.948 -0.561  0.148  0.384  … -1.000
1      6 -1.0 -1.0 -1.0 -1.000 -1.0 -1.000 -1.000 -1.000 -1.000  … -1.000
2      3 -1.0 -1.0 -1.0 -0.593  0.7  1.000  1.000  1.000  1.000  …  1.000
3      6 -1.0 -1.0 -1.0 -1.000 -1.0 -1.000 -1.000 -1.000 -1.000  … -1.000
4      6 -1.0 -1.0 -1.0 -1.000 -1.0 -1.000 -1.000 -0.858 -0.106  …  0.901

     248    249    250    251    252  253  254  255  256
0 -0.908  0.430  0.622 -0.973 -1.000 -1.0 -1.0 -1.0 -1.0
1 -1.000 -1.000 -1.000 -1.000 -1.000 -1.0 -1.0 -1.0 -1.0
2  0.717  0.333  0.162 -0.393 -1.000 -1.0 -1.0 -1.0 -1.0
3 -1.000 -1.000 -1.000 -1.000 -1.000 -1.0 -1.0 -1.0 -1.0
4  0.901  0.901  0.290 -0.369 -0.867 -1.0 -1.0 -1.0 -1.0

[5 rows x 257 columns]
```

[4]:
```
##ENSURE IMAGE CAN BE RECONSTRUCTED
##LOCATE THE RELEVANT ROW AND COLUMSN IN THE DF
check_im=train_df.loc[2, 1:]
##RESHAPE THE SERIES VALUES TO A MATRIX
im=check_im.values.reshape(16, 16)
plt.gray()
plt.imshow(im)

##this is Naredo code for numpy array
```

```
# 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
```

[4]: <matplotlib.image.AxesImage at 0x2416c8e40a0>



3.1 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).

3.2 Filter the training set

For the test and training sets:

Keep data where class == 2 or 3 only

Replace Class==2 with Class==-1

Replace Class==3 with Class==+1

As the below code is recurve if you have an error in the df management you need to reimport and run the previous cells. Other option would be to name the sets pulled from the dictionary something other than 'train_df' or 'test_df'.

[5]:
```
df_list=[train_df, test_df]

dic={}

for i, df in enumerate(df_list):

    ##LOCATE ALL ROWS WHERE CLASS==2 OR CLASS==3 AND RESET THE INDEX FOR THE␣
 ↪REDUCED DF
    new=df.loc[(df['CLASS']==2) | (df['CLASS']==3)].reset_index(drop=True).
 ↪copy()

    ##REPLACE CLASS 2 WITH -1
    new.loc[new['CLASS']==2, 'CLASS']=-1
    ##REPLACE CLASS 3 WITH +1
    new.loc[new['CLASS']==3, 'CLASS']=1

    dic[i]=new

###PULL THE MODIFIED DFS out of the dictionary
train_df=dic[0]
test_df=dic[1]

train_df.head(10)
```

[5]:

	CLASS	1	2	3	4	5	6	7	8	9	…	247	\
0	1	-1.0	-1.0	-1.0	-1.00	-1.000	-0.928	-0.204	0.751	0.466	…	0.466	
1	1	-1.0	-1.0	-1.0	-0.83	0.442	1.000	1.000	0.479	-0.328	…	1.000	
2	1	-1.0	-1.0	-1.0	-1.00	-1.000	-0.104	0.549	0.579	0.579	…	0.388	
3	1	-1.0	-1.0	-1.0	-1.00	-1.000	-1.000	-0.107	1.000	1.000	…	-0.280	
4	1	-1.0	-1.0	-1.0	-1.00	-0.674	0.492	0.573	0.755	-0.018	…	0.537	
5	-1	-1.0	-1.0	-1.0	-1.00	-1.000	-0.798	0.300	0.432	-0.799	…	-0.947	
6	1	-1.0	-1.0	-1.0	-1.00	0.330	0.522	0.693	1.000	0.786	…	0.617	
7	1	-1.0	-1.0	-1.0	-1.00	-1.000	-1.000	-0.805	0.503	0.591	…	0.314	
8	-1	-1.0	-1.0	-1.0	-1.00	-1.000	-0.638	0.222	0.706	1.000	…	-0.576	
9	-1	-1.0	-1.0	-1.0	-1.00	-1.000	-1.000	-1.000	-1.000	-1.000	…	-1.000	

	248	249	250	251	252	253	254	255	256
0	0.639	1.000	1.000	0.791	0.439	-0.199	-0.883	-1.000	-1.0
1	0.671	0.345	-0.507	-1.000	-1.000	-1.000	-1.000	-1.000	-1.0
2	0.579	0.811	1.000	1.000	0.715	0.107	-0.526	-1.000	-1.0
3	0.322	0.813	1.000	1.000	0.633	-0.144	-0.994	-1.000	-1.0
4	1.000	1.000	0.689	-0.530	-1.000	-1.000	-1.000	-1.000	-1.0
5	-0.524	0.307	0.390	0.852	0.751	0.990	0.567	-0.664	-1.0

```
6  1.000  0.976  0.469 -0.357 -0.975 -1.000 -1.000 -1.000 -1.0
7  0.590  0.842  0.487  0.062 -0.824 -1.000 -1.000 -1.000 -1.0
8  0.635  0.755  0.549  0.273  0.074 -0.083 -1.000 -1.000 -1.0
9 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.0

[10 rows x 257 columns]
```

4 Split dfs into target 'y' and data 'x' arrays

[6]:
```
##TRAINING DATA SPLIT
train_X=train_df.drop('CLASS', axis=1).values.copy()
train_y=train_df['CLASS'].values.copy()

print(train_X[0])

##TEST DATA SPLIT
test_X=test_df.drop('CLASS', axis=1).values.copy()
test_known_y=test_df['CLASS'].values.copy()
```

```
[-1.    -1.    -1.    -1.    -1.    -0.928 -0.204  0.751  0.466  0.234
 -0.809 -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.
 -0.37   0.739  1.     1.     1.     1.     0.644 -0.89  -1.    -1.
 -1.    -1.    -1.    -1.    -1.    -1.     0.616  1.     0.688 -0.455
 -0.731  0.659  1.    -0.287 -1.    -1.    -1.    -1.    -1.    -1.
 -1.    -1.    -0.376 -0.186 -0.874 -1.    -1.    -0.014  1.    -0.253
 -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.
 -1.    -1.    -0.978  0.501  1.    -0.54  -1.    -1.    -1.    -1.
 -1.    -1.    -1.    -1.    -1.    -0.998 -0.341  0.296  0.371  1.
  0.417 -0.989 -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.
 -1.    -0.008  1.     1.     1.     1.     0.761 -0.731 -1.    -1.
 -1.    -1.    -1.    -1.    -1.    -1.    -1.     0.242  1.     1.
  0.319  0.259  1.     0.742 -0.757 -1.    -1.    -1.    -1.    -1.
 -1.    -1.    -1.    -0.975 -0.467 -0.989 -1.    -1.    -0.171  0.998
  0.669 -0.945 -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.
 -1.    -1.    -1.    -1.    -1.     0.228  1.     0.038 -1.    -1.
 -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.
 -1.    -0.826  0.918  0.933 -0.794 -1.    -1.    -1.    -0.666  0.337
  0.224 -0.908 -1.    -1.    -1.    -1.    -1.    -1.     0.418  1.
 -0.258 -1.    -1.    -0.246  1.     1.     0.355 -0.958 -1.    -1.
 -1.    -1.    -1.    -1.    -0.077  1.     0.344 -1.    -1.     0.075
  1.     1.     0.649  0.256 -0.2   -0.351 -0.733 -0.733 -0.733 -0.433
  0.649  1.     0.093 -1.    -1.    -0.959 -0.062  0.821  1.     1.
  1.     1.     1.     1.     1.     1.     1.     0.583 -0.843 -1.
 -1.    -1.    -1.    -0.877 -0.326  0.174  0.466  0.639  1.     1.
  0.791  0.439 -0.199 -0.883 -1.    -1.   ]
```

4.1 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:

[7]:
```
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_
```

5 Define a function to calculate the distance between two points

[8]:
```
def get_distance(arr1, arr2):

    import numpy as np

    ##GET THE DIFFERENCE FOR EVERY ELEMENT IN BOTH ARRAYS, SQ VALUES, SUM␣
 ↪ARRAY, GET SQRT
    dist=np.sqrt(np.sum((arr1-arr2)**2))

    return dist


##CODE TO CONFIRM OPERATING AS EXPECTED
# arrx=np.arange(1, 5, 1)
# arry=np.arange(2, 10, 2)

# a=get_distance(arry, arrx)

# print(arrx, arry)
# print(a)
```

6 Define a function to compare single test point to all known data points

This function will take a single test point and the training dataset. For every point in the training dataset, it will calculate the distance using the previously defined distance function. The output of this function is vector of distance values comparing the single point to the training dataset

[9]:
```
def compare_test_point(test_row, training_dataset):

    import numpy as np

    ##CREATE AN ZEROS VECTOR WITH ROWS MATCHING LENGTH OF TRAINING DATASET
    dist_vector=np.zeros((len(training_dataset), 1))
#     print('ZERO', dist_vector)

    ##FOR EVERY ROW OF THE TRAINING DATA
    for i, rw in enumerate(training_dataset):
        ##CALCULATE THE DISTANCE AND APPEND IT TO THE DISTANCE VECTOR ARRAY
        dist_vector[i]=get_distance(test_row, rw)

    return dist_vector

# ##TEST WHAT'S HAPPENING
# ##GET SINGLE EXAMPLE
# a=train_X[5]
# ##MATRIX OF 10 ROWS OF TRAINING 'X' DATA
# b=train_X[0:10]
# ##RUN TEST
# c=compare_test_point(a, b)
# ##IF TEST IS ACCURATE THE TRAINING EXAMPLE SHOULD BE ZERO AT THE ROW NUMBER␣
 ↪IN A
# print(c)
```

7 Find the 'x' closest matches

This function takes the distance vector calculated in the previous step. Turn it into a pandas series index from 0-len(vector). Sort the series by smallest to largest. Using the indexes for the smallest distances, access the Target data at those indexes

[10]:
```
def find_nearest(test_example, train_X, train_y, num_nearest):

    import numpy as np
    import pandas as pd

    ###CALL DIST VECTOR FOR A GIVEN EXAMPLE
    dist_arr=compare_test_point(test_example, train_X)
#     print(dist_arr)
```

```
    ser=pd.Series(dist_arr.flatten()) ##series defaults to range index
#     print('CONVERT TO SER', ser)
    ser.sort_values(axis=0, ascending=True, inplace=True) ##Sort the series
#     print('SORTED SER', ser)
    ind=ser.iloc[0:num_nearest].index ##return index values of the 'x' nearest␣
 ↪neighbours

    ##GET X CLOSEST TRAINING CLASSIFICATION
    training_class=train_y[ind].tolist()
#     print('TRAINING CLASS', training_class, type(training_class))


    ##GET THE MAJORITY WINNER (MOST FREQUENT OCCURRING ITEM)
    majority = max(set(training_class), key=training_class.count)
#     print('MAJORITY', majority)

    ##Using the target data find the 'x' nearest classifications
    return majority

# ##TEST WHAT'S HAPPENING
# ##WE KNOW THE CLASS IS 1 AT ROW 0 (ACCURATELY CONFIRMED) AND -1 AT ROW 5␣
 ↪(INACCURATE RETURNS 1)
# ##GET SINGLE EXAMPLE
# a=train_X[5]
# ##MATRIX OF 10 ROWS OF TRAINING 'X' DATA
# b=train_X[0:10]

# ##RUN TEST FOR 1 NEAREST NEIGHBOUR
# c=find_nearest(a, b, train_y, 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.

8 Implement K-means for entire test set with k=1

For every row of the test_X get a majority winner (i.e. estimate, based on training set, what the classification is)

[11]:
```
##CREATE A COPY OF THE TEST DF
df=test_df.copy()
##ADD A LOCATION WHERE WE CAN KEEP THE ESTIMATED CLASS
df['ESTIMATED_CLASS']=np.nan

##FOR EVERY ROW OF THE DF
for i, rw in df.iterrows():
    rw.drop(['CLASS', 'ESTIMATED_CLASS'], axis=0, inplace=True)

    df.loc[i, 'ESTIMATED_CLASS']=find_nearest(rw.values, train_X, train_y, 1)

##CHANGE DATATYPE OF ESTIMATED COLUMN TO INTEGER
df['ESTIMATED_CLASS']=df['ESTIMATED_CLASS'].astype(int)
df.head(6)
```

[11]:

	CLASS	1	2	3	4	5	6	7	8	9	…	\
0	1	-1.000	-1.000	-1.000	-0.593	0.700	1.000	1.000	1.000	1.000	…	
1	-1	-0.996	0.572	0.396	0.063	-0.506	-0.847	-1.000	-1.000	-1.000	…	
2	-1	-1.000	-1.000	0.469	0.413	1.000	1.000	0.462	-0.116	-0.937	…	
3	1	-1.000	-1.000	-1.000	0.264	0.532	-0.210	-0.746	-0.779	-1.000	…	
4	-1	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	…	
5	-1	-1.000	-1.000	-1.000	-0.831	0.047	0.140	0.947	0.813	0.012	…	

	248	249	250	251	252	253	254	255	256	\
0	0.717	0.333	0.162	-0.393	-1.000	-1.000	-1.000	-1.000	-1.000	
1	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	
2	1.000	1.000	0.270	-0.280	-0.855	-1.000	-1.000	-1.000	-1.000	
3	0.418	-0.057	-0.829	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	
4	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	
5	-1.000	-1.000	-1.000	-1.000	-0.866	-0.672	0.131	0.135	-0.318	

	ESTIMATED_CLASS
0	1
1	-1
2	-1
3	1
4	-1
5	-1

[6 rows x 258 columns]

9 Implement K-means for entire test set with multiple 'k's'

Hold output in a dictionary where key is the number of nearest neighbours used in the majority analysis. Data will be held in tupled list (row index, K classification)

[12]:
```
##CREATE A COPY OF THE TEST DF
df=test_df.copy()

##ADD A LOCATION WHERE WE CAN KEEP THE ESTIMATED CLASS
##KEY IS NUM NEAREST NEIGHBOURS VALUE
##VALUE IS INDEX AND ESTIMATED CLASS IN
class_dic={}

##seems to be taking a lot of time lets check time to run
start_time=time.time()

for k in range(1, 31, 1):
    dic={}
    ##FOR EVERY ROW OF THE DF
    for i, rw in df.iterrows():
        rw.drop(['CLASS'], axis=0, inplace=True)

        dic[i] = find_nearest(rw.values, train_X, train_y, k)
    ##ADD INNER DIC TO DICTIONARY
    class_dic[k]=dic

print('cell takes %s minutes to run' %((time.time()-start_time)/60))
```

```
cell takes 3.5854053695996604 minutes to run
```

[13]:
```
df=pd.DataFrame(class_dic)
k_num=df.columns
mux=pd.MultiIndex.from_product([['KNN_NUM'], df.columns])
df.columns=mux

df
```

[13]:

	KNN_NUM										…										
	1	2	3	4	5	6	7	8	9	10	…	21	22	23	24	25	26	27	28	29	30
0	1	1	1	1	1	1	1	1	1	1	…	1	1	1	1	1	1	1	1	1	1
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	…	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	1	-1	1	1	1	1	1	…	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	…	1	1	1	1	1	1	1	1	1	1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	…	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
..	…	..	..	..	..	..	..	..	..	..	…	..	..	..	..	..	..	..	..	..	..
359	1	1	1	1	1	1	1	1	1	1	…	1	1	1	1	1	1	1	1	1	1
360	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	…	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
361	1	1	1	1	1	1	1	1	1	1	…	1	1	1	1	1	1	1	1	1	1

```
362      -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  … -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
363       1  1  1  1  1  1  1  1  1  1  …  1  1  1  1  1  1  1  1  1  1

[364 rows x 30 columns]
```

10 Check the accuracy of K-means for each K-num

Here we use sklearn to calculate the accuracy of each of the classifiers.

10.1 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?

[14]:
```
acc_df=df.copy()

accuracy_dic={}
for i, col in acc_df.iteritems():

    ##GET THE COLUMN VALUES
    val=col.values
    ##CALCULATE ACCURACY AGAINST THE KNOWN CLASS
    accuracy_dic[i[1]]=accuracy_score(test_known_y, col)

# accuracy_dic
```

[26]:
```
acc_df=pd.DataFrame(accuracy_dic.values(), index=accuracy_dic.keys())
acc_df
```

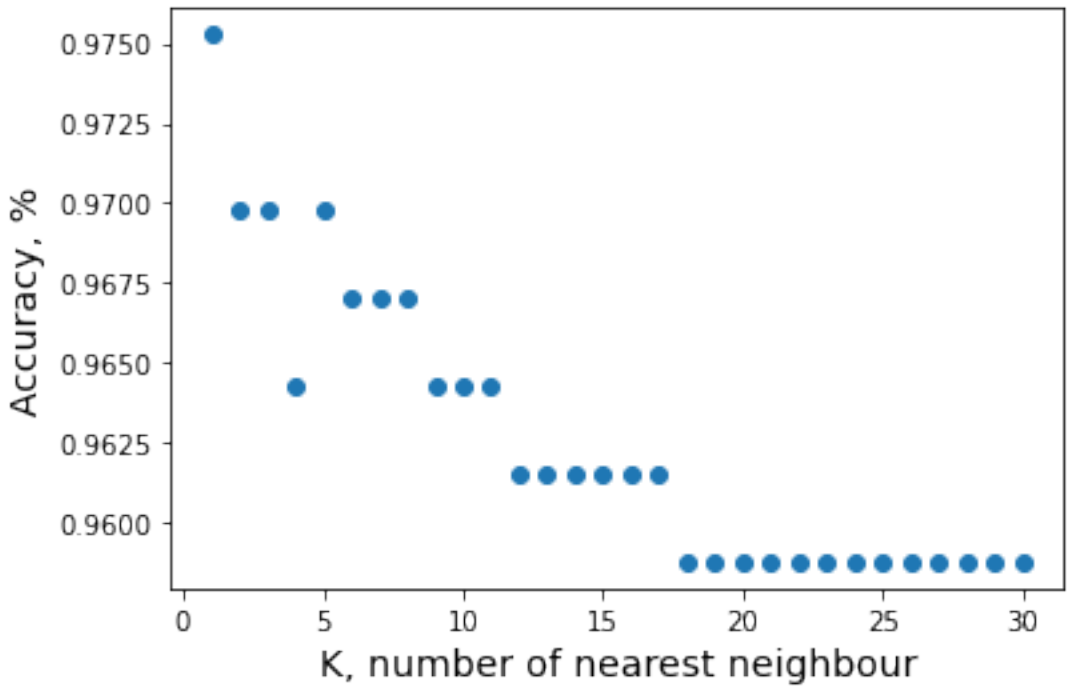
[26]:

	0
1	0.975275
2	0.969780
3	0.969780
4	0.964286
5	0.969780
6	0.967033
7	0.967033
8	0.967033
9	0.964286
10	0.964286
11	0.964286
12	0.961538
13	0.961538
14	0.961538

```
15  0.961538
16  0.961538
17  0.961538
18  0.958791
19  0.958791
20  0.958791
21  0.958791
22  0.958791
23  0.958791
24  0.958791
25  0.958791
26  0.958791
27  0.958791
28  0.958791
29  0.958791
30  0.958791
```

[34]:
```
x=acc_df.index
y=acc_df.values

plt.scatter(x, y)
plt.xlabel('K, number of nearest neighbour', fontsize=14)
plt.ylabel('Accuracy, %', fontsize=14)
```

[34]:
```
Text(0, 0.5, 'Accuracy, %')
```



11 What happens if we don't filter the data

Can we apply this version of K means to larger datasets? How does this affect accuracy?

[50]:
```
##REIMPORT THE DATA AS IT WAS ORIGINALLY PROVIDED

##LOAD DATA INTO NUMPY ARRAY
train_data = np.loadtxt("mnist.train")
test_data  = np.loadtxt("mnist.test")

##CHECK SIZE OF THE ARRAYS
print(test_data.shape, test_data.shape)

##FOR DF WE NEED COL HEADINGS USE INT VALUES TO GIVE DATA HEADINGS
##FIRST COL IS ACTUALLY THE CLASSIFICATION SO WE'LL GIVE THAT A NAME NOT A␣
 ↪NUMBER
col_head=[i for i in range(1, train_data.shape[1], 1)]
##ADD IN CLASS HEADING FOR THE FIRST COLUMM
col_head.insert(0, 'CLASS')

##CONVERT TO DF
train_df=pd.DataFrame(train_data, columns=col_head)
test_df=pd.DataFrame(test_data, columns=col_head)

##ENSURE CLASS COLUMN IS INTEGER TYPE DATA
train_df['CLASS']=train_df['CLASS'].astype(int)
test_df['CLASS']=test_df['CLASS'].astype(int)
```

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

[51]:
```
test_X=test_df.drop('CLASS', axis=1).values
train_X=train_df.drop('CLASS', axis=1).values
train_y=train_df['CLASS'].values
```

[52]:
```
##ADD A LOCATION WHERE WE CAN KEEP THE ESTIMATED CLASS
##KEY IS NUM NEAREST NEIGHBOURS VALUE
##VALUE IS INDEX AND ESTIMATED CLASS IN
class_dic={}

total_start=time.time()

for k in range(1, 11, 1):
    ##seems to be taking a lot of time lets check time to run
    k_time=time.time()
```

```
    dic={}
    ##FOR EVERY ROW OF THE DF
    for i, rw in enumerate(test_X):

        dic[i] = find_nearest(rw, train_X, train_y, k)
    ##ADD INNER DIC TO DICTIONARY
    class_dic[k]=dic

    print('K number ', k)
    print('K number takes %s minutes to run' %((time.time()-k_time)/60))

print('cell takes %s minutes to run' %((time.time()-total_start)/60))
```

```
K number  1
K number takes 10.854120516777039 minutes to run
K number  2
K number takes 15.283987851937612 minutes to run
K number  3
K number takes 20.235027492046356 minutes to run
K number  4
K number takes 24.85471364657084 minutes to run
K number  5
K number takes 29.077772136529287 minutes to run
K number  6
K number takes 33.32473753690719 minutes to run
K number  7
K number takes 37.84116010268529 minutes to run
K number  8
K number takes 42.23450837532679 minutes to run
K number  9
K number takes 46.80264442761739 minutes to run
K number  10
K number takes 51.342769809563954 minutes to run
cell takes 51.34281977415085 minutes to run
```

[59]:
```
df=pd.DataFrame(class_dic)
k_num=df.columns
mux=pd.MultiIndex.from_product([['KNN_NUM'], df.columns])
df.columns=mux

estimated_class=df.copy()
estimated_class
```

[59]:

	KNN_NUM									
	1	2	3	4	5	6	7	8	9	10
0	9	9	9	9	9	9	9	9	9	9
1	6	6	6	6	6	6	6	6	6	6

```
2          3  3  3  3  3  3  3  3  3  3
3          6  6  6  0  6  6  6  6  6  6
4          6  6  6  6  6  6  6  6  6  6
…       … .. .. .. .. .. .. .. .. ..
2002       3  3  3  3  3  3  3  3  3  3
2003       9  9  9  9  9  9  9  9  9  9
2004       9  9  4  4  4  4  4  4  4  4
2005       0  0  0  0  0  0  0  0  0  0
2006       1  1  1  1  1  1  1  1  1  1

[2007 rows x 10 columns]
```

[60]:
```
acc_df=df.copy()

accuracy_dic={}
for i, col in acc_df.iteritems():

    ##GET THE COLUMN VALUES
    val=col.values
    ##CALCULATE ACCURACY AGAINST THE KNOWN CLASS
    accuracy_dic[i[1]]=accuracy_score(test_df['CLASS'].values, col)
```

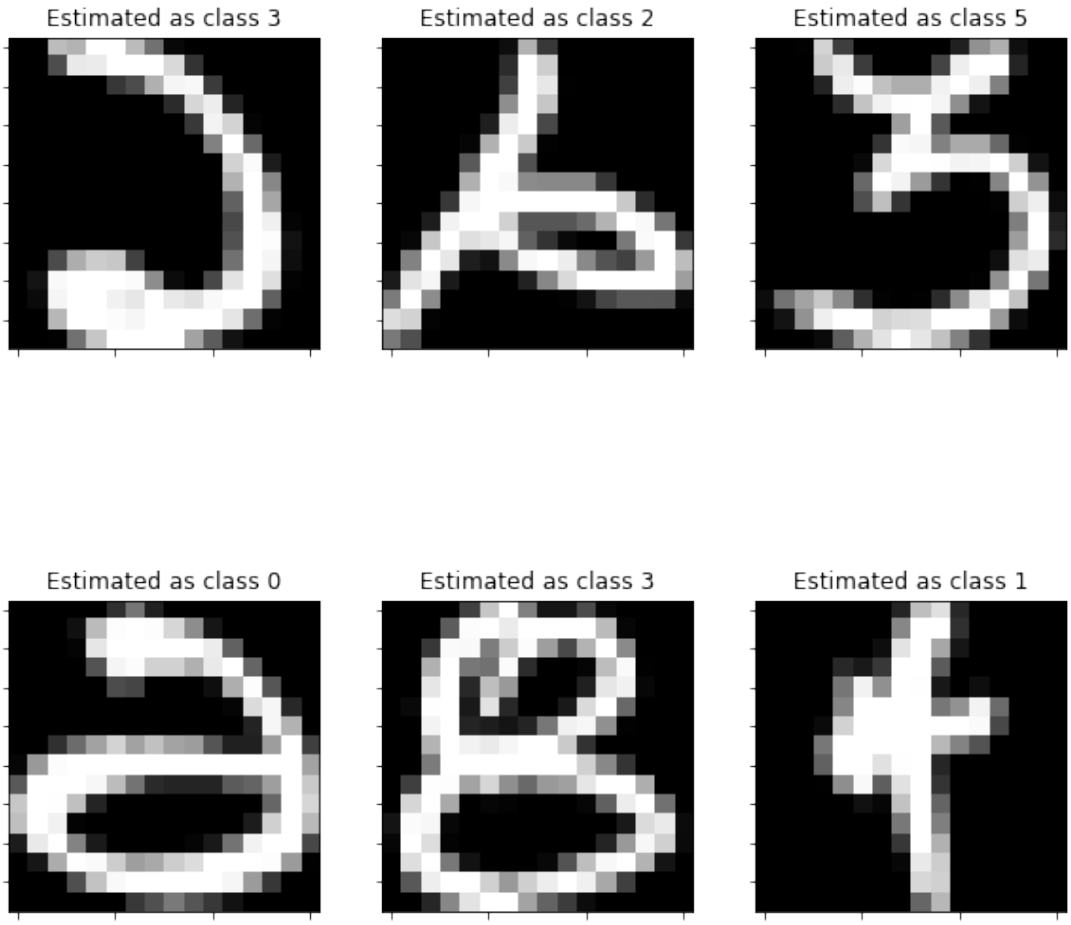
[65]:
```
acc_df=pd.DataFrame(accuracy_dic.values(), index=accuracy_dic.keys())
acc_df.columns=['accuracy']
acc_df
```

[65]:
```
    accuracy
1   0.943697
2   0.931739
3   0.944694
4   0.943697
5   0.943199
6   0.940708
7   0.941704
8   0.939711
9   0.939711
10  0.936223
```

[131]:
```
##Combine accuracy df with original classification (test_y)
test_y=test_df['CLASS']
dfx=estimated_class.copy().droplevel(level=0, axis=1)

dfy=pd.concat([dfx, test_y], axis=1)

# print(dfy.columns)

##FIND CLASSIFICATIONS THAT DON'T MATCH (SEARCH K=1 COMPARED TO KNOWN CLASS)
```

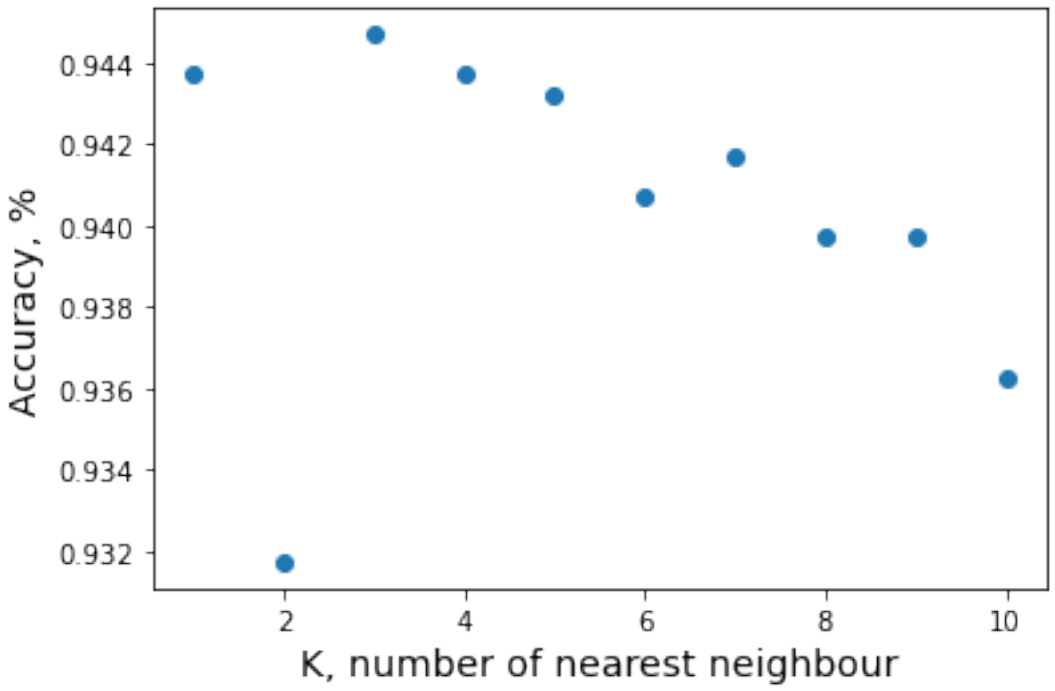
```
dfy['DIFF']=dfy[10]-dfy['CLASS']
wrong=dfy.loc[dfy['DIFF']!=0]

print('%s examples were incorrectly classified for K=10' %(len(wrong)))

print('Here I plot the first 6 incorrectly classified examples')

wrong_five=wrong.iloc[0:6, :]
index_pos=wrong_five.index.to_list()
print(index_pos)

plt.figure(figsize=(10,10))

for i, val in enumerate(index_pos, 1):
#     print('Actual class %d' %(wrong_five.loc[val, 'CLASS']))
    plt.subplot(2, 3, i)
    img=test_df.iloc[val].drop('CLASS').values.reshape(16,16)
    plt.gray()
    plt.imshow(img)
    plt.xticks(color='w')
    plt.yticks(color='w')
    plt.title('Estimated as class %s' %(wrong_five.loc[val, 10]))
```

```
128 examples were incorrectly classified for K=10
Here I plot the first 6 incorrectly classified examples
[12, 17, 27, 53, 87, 92]
```



[57]:
```
##Plot accuracy v. K
x=acc_df.index
y=acc_df.values

plt.scatter(x, y)
plt.xlabel('K, number of nearest neighbour', fontsize=14)
plt.ylabel('Accuracy, %', fontsize=14)
```

[57]:
```
Text(0, 0.5, 'Accuracy, %')
```



11.1 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?

Extending the algorithm to a multi-class setting slowed down the code considerably. This is not unexpected given we have also increased the number of test/training examples by including all the data not just a filtered section. For filtered data the best accuracy was achieved when K=1. The most accurate result for non-filted data however was found to be K=3.

These results are as expected when one considers that the filtered data looked only at two examples and those examples were fitted to a -1 or 1 relationship. Conversely the non-filtered data must account for significantly more classification possibilities. Considering this K=3 is the more reasonable approximation should this method be employed in a real-world senario.

- Create a plot of accuracy vs. K.

This has been completed for both the filtered and non-filtered data. Plots are shown in cells previous

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

This has been completed for both the non-filtered data. Plots are shown in cells previous

Analysis Questions

- What values of k did you try?

For filtered data we looked at K ranging from 1 to 30. For non-filtered data, due to the size of the dataset and the time required to run code we looked only at K ranging from 1 to 10.

- Which value of k produced the highest accuracy? What general trends did you observe as k increased?

If we considered the non-filtered data we can see that the most accurate results were found for K=3. With larger values of K resulting in a decrease in accuarcy. This makes sense in that if we use more than three nearest neighbours the Euclidean distance between comparative matches is ever increasing. Hence at K>3 the Euclidean distance is diverging further and further from the 'best match' and the majority selection tool is more likely to choose a classification different to the actual class.

In the filtered data we achieved the best accuracy for K=1. These results are as expected given that the filtered data is classifying each of the two possible classes as either '2' or '3' (or in this case '-1' and '1') so it's a simple case of it is class '2' or not.

- 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.

Extending the algorithm to a multi-class setting slowed down the code considerably. For the filtered dataset the sequence runtime for K=1 to 30 was approx 4 minutes. When we use the entire dataset (non-filtered data) the run time for K=1 was 10 minutes and each subsequent K approx 5 min. To run the range K=1 to 10 took just shy of 1 hour.
Given that we have performed separate functions for each calculation we must use a loop system to analyse each test point, loops are slower than vectorised calculations. We could speed the code up considerably by vectorising the comparison between the test dataset and the training dataset, though I lack clarity on how that could be implemented. (Other than using a sklearn process for example).

11.2 In depth questions

- If you are familiar with confusion matrices, create one for this test dataset and your "best" value of K.

Here we attempt a confusion matrix for the non-filtered data with K=10 (this was the least accurate so we have more 'misclassifications'

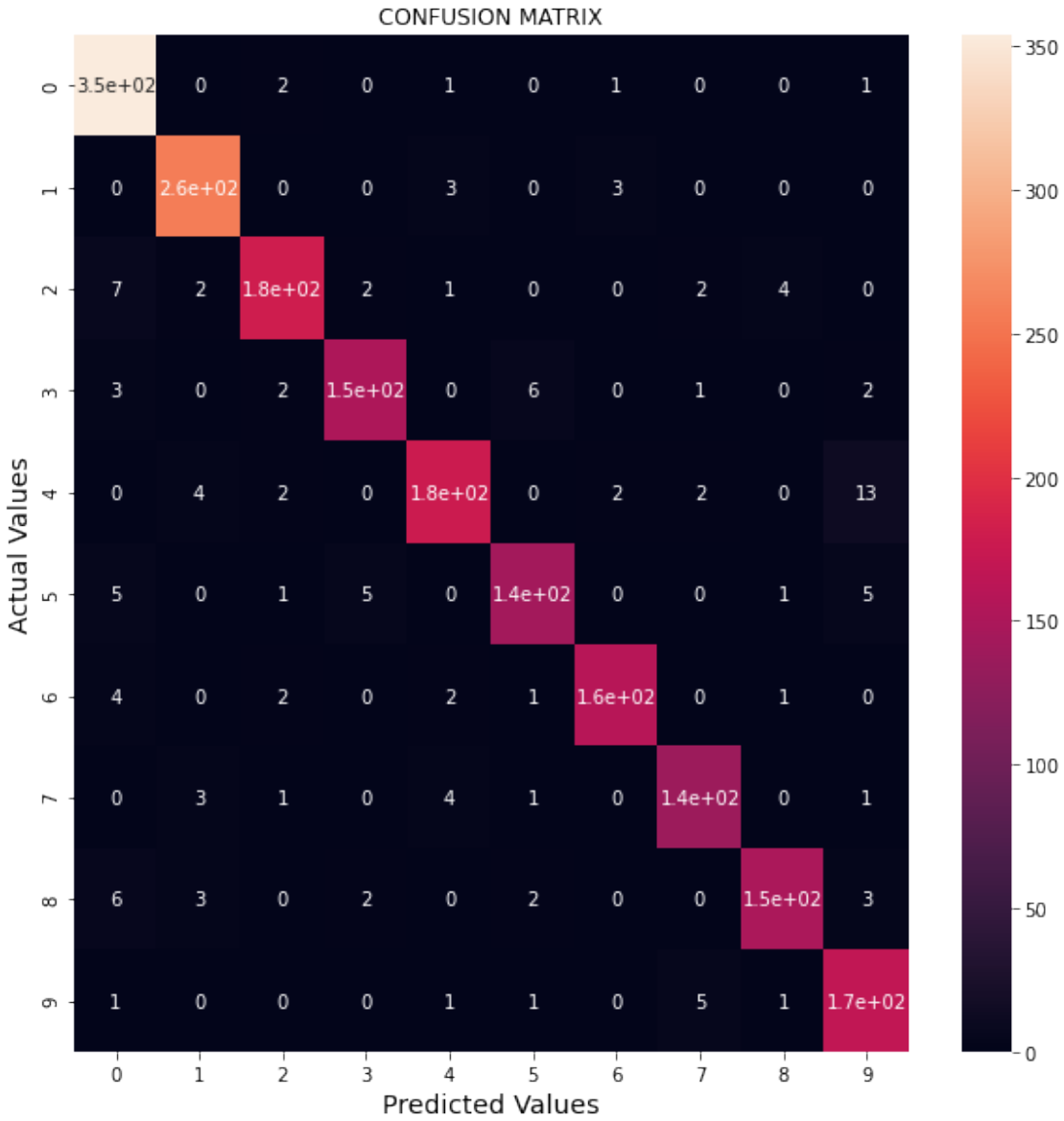
[148]:
```
##WE ALREADY HAVE THE GUESS AT K=10 AND THE KNOWN CLASS
df=dfy.copy()
df.drop([1, 2, 3, 4, 5, 6, 7, 8, 9, 'DIFF'], axis=1, inplace=True)

kten_pred=df[10]
known_class=df['CLASS']

from sklearn.metrics import confusion_matrix

cm=confusion_matrix(known_class, kten_pred)

cm_df=pd.DataFrame(cm)
```

```
# multi_col=pd.MultiIndex.from_product([['PREDICTED VALUES'], cm_df.columns])
# multi_ind=pd.MultiIndex.from_product([['ACTUAL VALUES'], cm_df.index])

# cm_df.columns=multi_col
# cm_df.index=multi_ind

cm_df
```

[148]:

	0	1	2	3	4	5	6	7	8	9
0	354	0	2	0	1	0	1	0	0	1
1	0	258	0	0	3	0	3	0	0	0
2	7	2	180	2	1	0	0	2	4	0
3	3	0	2	152	0	6	0	1	0	2
4	0	4	2	0	177	0	2	2	0	13
5	5	0	1	5	0	143	0	0	1	5
6	4	0	2	0	2	1	160	0	1	0
7	0	3	1	0	4	1	0	137	0	1
8	6	3	0	2	0	2	0	0	150	3
9	1	0	0	0	1	1	0	5	1	168

[152]:
```
##PLOT HEAT MAP
import seaborn as sns

plt.figure(figsize=(10,10))
sns.heatmap(cm_df, annot=True)
plt.title('CONFUSION MATRIX')
plt.ylabel('Actual Values', fontsize=14)
plt.xlabel('Predicted Values', fontsize=14)

plt.show()
```


CONFUSION MATRIX
Actual Values
Predicted Values
0 1 2 3 4 5 6 7 8 9
3.5e+02 0 2 0 1 0 1 0 0 1
0 2.6e+02 0 0 3 0 3 0 0 0
7 2 1.8e+02 2 1 0 0 2 4 0
3 0 2 1.5e+02 0 6 0 1 0 2
0 4 2 0 1.8e+02 0 2 2 0 13
5 0 1 5 0 1.4e+02 0 0 1 5
4 0 2 0 2 1 1.6e+02 0 1 0
0 3 1 0 4 1 0 1.4e+02 0 1
6 3 0 2 0 2 0 0 1.5e+02 3
1 0 0 0 1 1 0 5 1 1.7e+02
350 300 250 200 150 100 50 0

[ ]: