**Functions:**

These functions are to calculate the distance between 2 vectors:

def vector\_subtract(v, w):

return [v\_i - w\_i for v\_i, w\_i in zip(v,w)]

def sum\_of\_squares(v):

return sum(v\_i \* v\_i for v\_i in v)

def distance(v, w):

s = vector\_subtract(v,w)

return math.sqrt(sum\_of\_squares(s))

This function is to build a dictionary from the txt file:

def new\_dict(a):

a\_dict=dict()

with open (a) as file\_object:

small\_count=0

huge\_count=0

for line in file\_object:

small\_count=small\_count+1

if small\_count==1:

vector\_list=list()

if len(line)>7:

line\_list=list()

for i in line:

if i.isdigit():

line\_list.append(int(i))

vector\_list.append(tuple(line\_list))

if len(line)<7:

small\_count=0

huge\_count=huge\_count+1

for i in line:

if i.isdigit():

a\_dict[i+'('+str(huge\_count)+')']=vector\_list

return a\_dict

I have 2 kinds of functions to predict the possible number. The first is not optimized to reducing the kNN computation time. It will search all the vectors in the training dictionary and compute distance between the testing vector and all the training vectors and find the best k neighbors. Because of the computer always mistakes 4 and 9, then I discover that the main difference between the 2 numbers is focus on the first half part. Therefore, I adjust the wight of the 2 half parts the first half part is more important, I multiply it 1.2 times and multiply the second half 0.8 times. This method improves the accuracy. The parameter k represents the value of k. This kind of method have high accuracy about 96% but it cost a many time about 35s in my computer (i5-7300HQ).

def predict(m,k):

distance\_dict=dict()

for number\_training,vector\_training in training\_dict.items():

difference=0

for i in range(0,len(vector\_training)):

if i<0.5\*len(vector\_training):

difference+=distance(m[i],vector\_training[i])\*1.2

else:

difference+=distance(m[i],vector\_training[i])\*0.8

distance\_dict[number\_training]=difference

distance\_list=list()

for key,value in distance\_dict.items():

distance\_list.append((value,key))

distance\_list.sort()

alternative\_list=list()

for i in range(k):

alternative\_list.append(distance\_list[i][1][0])

alternative\_list\_counts=Counter(alternative\_list)

result=alternative\_list\_counts.most\_common(1)[0][0]

return result

The second method use random method to optimize in reducing the kNN computation time. In my function, I randomly pick 15 samples for every number from the training dictionary. The accuracy is about 92%, but it’s more efficient, the time is about 6 s in my computer.

def predict(m,k):

distance\_dict=dict()

sample\_list=[[],[],[],[],[],[],[],[],[],[]]

sample\_dict=dict()

for key,value in training\_dict.items():

for i in range(10):

if int(key[0])==i:

sample\_list[i].append(value)

count=0

for i in sample\_list:

sample\_list[count]=random.sample(sample\_list[count],15)

count+=1

count=0

for i in range(10):

for j in range(15):

count+=1

sample\_dict[str(i)+'('+str(count)+')']=sample\_list[i][j]

for number\_training,vector\_training in sample\_dict.items():

difference=0

for i in range(0,len(vector\_training)):

if i<0.5\*len(vector\_training):

difference+=distance(m[i],vector\_training[i])\*1.2

else:

difference+=distance(m[i],vector\_training[i])\*0.8

distance\_dict[number\_training]=difference

distance\_list=list()

for key,value in distance\_dict.items():

distance\_list.append((value,key))

distance\_list.sort()

alternative\_list=list()

for i in range(k):

alternative\_list.append(distance\_list[i][1][0])

alternative\_list\_counts=Counter(alternative\_list)

result=alternative\_list\_counts.most\_common(1)[0][0]

return result

The next function is for timing:

from time import strftime,gmtime,localtime

starttime=strftime("%Y-%m-%d %H:%M:%S", localtime())

endtime=strftime("%Y-%m-%d %H:%M:%S", localtime())

The next function is for find the nearest k neighbors:

from collections import Counter

for i in range(k):

alternative\_list.append(distance\_list[i][1][0])

alternative\_list\_counts=Counter(alternative\_list)

result=alternative\_list\_counts.most\_common(1)[0][0]

**variables and program flow:**

Firstly, I build some dictionary to restore the information from the txt file. Predicting dictionary is to store the result of the predict, which will be used to calculate the accuracy and statistics:

training='digit-training.txt'

testing='digit-testing.txt'

predicting='digit-predict.txt'

training\_dict=new\_dict(training)

testing\_dict=new\_dict(testing)

predict\_dict=dict()

I build some list to prepare for storing the result of the accuracy, correct, mistake, total number statistics:

training\_count\_list=[0,0,0,0,0,0,0,0,0,0]

testing\_count\_list=[0,0,0,0,0,0,0,0,0,0]

correct\_count\_list=[0,0,0,0,0,0,0,0,0,0]

mistake\_count\_list=[0,0,0,0,0,0,0,0,0,0]

accuracy\_list=[0,0,0,0,0,0,0,0,0,0]

number\_list=[0,1,2,3,4,5,6,7,8,9]

This flow is to store the result of the total number statistics from the training and testing dictionary:

for key in training\_dict:

for i in number\_list:

if int(key[0])==i:

training\_count\_list[i]+=1

for key in testing\_dict:

for i in number\_list:

if int(key[0])==i:

testing\_count\_list[i]+=1

This flow is to store the result of the prediction for the number from testing dictionary in predicting dictionary:

for number\_testing,vector\_testing in testing\_dict.items():

predict\_dict[number\_testing]=predict(vector\_testing,5)

These flows is to store the results of the statistics of the correct, mistake, accuracy in the lists I mentioned before:

for key,value in predict\_dict.items():

if not key[0]==value:

mistake\_count\_list[int(key[0])]+=1

for key,value in predict\_dict.items():

if key[0]==value:

correct\_count\_list[int(key[0])]+=1

for i in range(10):

accuracy\_list[i]=correct\_count\_list[i]/testing\_count\_list[i]

This flow is to output the information of the result of the training and testing process:

print('Begin of Training @'+starttime+'\n')

print('----------------------------------------\n')

print(' Training Info '+'\n')

print('----------------------------------------\n')

for i in range(10):

print(' '+str(i)+' = '+str(training\_count\_list[i])+'\n')

print('----------------------------------------\n')

print(' Total Samples = '+str(sum(training\_count\_list))+'\n')

print('----------------------------------------\n')

print(' Testing Info '+'\n')

print('----------------------------------------\n')

for i in range(10):

print(' '+str(i)+' = '+str(correct\_count\_list[i])+', '+str(mistake\_count\_list[i])+', '+str(int(accuracy\_list[i]\*100))+'%'+'\n')

print('----------------------------------------\n')

print(' Accuracy = '+str(round(sum(correct\_count\_list)/sum(testing\_count\_list),2)\*100)+'%'+'\n')

print(' Correct/Total = '+str(sum(correct\_count\_list))+'/'+str(sum(testing\_count\_list))+'\n')

print('----------------------------------------\n')

print('End of Training @'+endtime+'\n')

This flow is to output the result of the prediction of the predict txt file:

for vector\_predicting in new\_dict(predicting).values():

print(predict(vector\_predicting,3))

**kNN model:**

a. the rule(s) used in making the prediction:

My method is to compute the distance of the corresponding vector and sum them up for ranking. For example, a is [(1,2,3), (2,3,4)], b is [(2,3,4), (5,6,7)], so the result is the total distance = distance((1,2,3),(2,3,4))+ distance((5,6,7),(2,3,4)) .Using for loop, I calculate distance between the testing vector and all the training vectors and store all the result in the predict dictionary and rank them. Because of the computer always mistakes 4 and 9, then I discover that the main difference between the 2 numbers is focus on the first half part. Therefore, I adjust the wight of the 2 half parts the first half part is more important, I multiply it 1.2 times and multiply the second half 0.8 times. This method improves the accuracy.

def predict(m,k):

distance\_dict=dict()

for number\_training,vector\_training in training\_dict.items():

difference=0

for i in range(0,len(vector\_training)):

if i<0.5\*len(vector\_training):

difference+=distance(m[i],vector\_training[i])\*1.2

else:

difference+=distance(m[i],vector\_training[i])\*0.8

distance\_dict[number\_training]=difference

distance\_list=list()

for key,value in distance\_dict.items():

distance\_list.append((value,key))

distance\_list.sort()

alternative\_list=list()

for i in range(k):

alternative\_list.append(distance\_list[i][1][0])

alternative\_list\_counts=Counter(alternative\_list)

result=alternative\_list\_counts.most\_common(1)[0][0]

return result

For the optimized model, I use random method to optimize in reducing the kNN computation time. In my function, I randomly pick 15 samples for every number from the training dictionary.

def predict(m,k):

distance\_dict=dict()

sample\_list=[[],[],[],[],[],[],[],[],[],[]]

sample\_dict=dict()

for key,value in training\_dict.items():

for i in range(10):

if int(key[0])==i:

sample\_list[i].append(value)

count=0

for i in sample\_list:

sample\_list[count]=random.sample(sample\_list[count],15)

count+=1

count=0

for i in range(10):

for j in range(15):

count+=1

sample\_dict[str(i)+'('+str(count)+')']=sample\_list[i][j]

for number\_training,vector\_training in sample\_dict.items():

difference=0

for i in range(0,len(vector\_training)):

if i<0.5\*len(vector\_training):

difference+=distance(m[i],vector\_training[i])\*1.2

else:

difference+=distance(m[i],vector\_training[i])\*0.8

distance\_dict[number\_training]=difference

distance\_list=list()

for key,value in distance\_dict.items():

distance\_list.append((value,key))

distance\_list.sort()

alternative\_list=list()

for i in range(k):

alternative\_list.append(distance\_list[i][1][0])

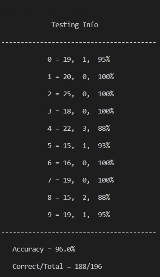
alternative\_list\_counts=Counter(alternative\_list)

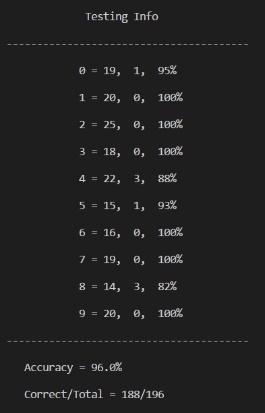
result=alternative\_list\_counts.most\_common(1)[0][0]

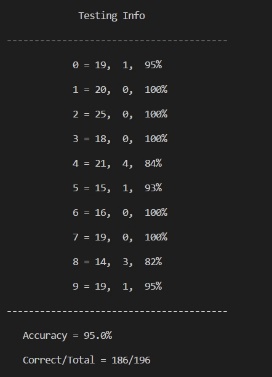
return result

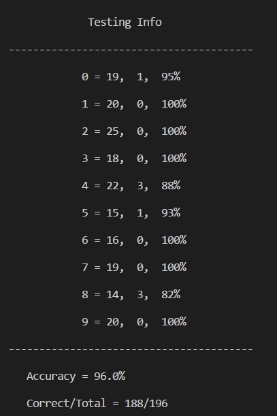
b. how the closest neighbors are determined

I have tested 3, 5, 7, 9. 3, 5, 9 all achieve the highest accuracy about 96%, and 5 and 9 get the most 100%. For 5 is smaller than 9, it will be more stable, so I choose 5.

3:

5:

7:

9:

c. one strategy in reducing the kNN computation time:

In my optimized version, I have used random method.

I think that it’s not all the vectors stored in the dictionary is useful. For example:

00000000000100000000000000000000

00000000001111100000000000000000

00000000011111111111110000000000

00000000011111111111111000000000

00000000111111111111111100000000

00000000111111111111111100000000

00000001111111111111111100000000

00000001111111001111111000000000

00000001111110000000000000000000

00000001111111000000000000000000

00000011111111110000000000000000

00000011111111111100000000000000

00000011111111111100000000000000

00000111111111111110000000000000

00000011111111111111000000000000

00000001111111111111110000000000

00000001111000001111110000000000

00000000000000000111110000000000

00000000000000000111110000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000111111000000000

00000000000000001111100000000000

00000000000000001111110000000000

00000000001111111111110000000000

00000000011111111111100000000000

00000000011111111111000000000000

00000000011111111110000000000000

00000000001111111110000000000000

00000000000011100000000000000000

Represents 5. However, if I reduce the first 3 lines and the last 3 lines, I can still recognize that it is 5:

00000000011111111111111000000000

00000000111111111111111100000000

00000000111111111111111100000000

00000001111111111111111100000000

00000001111111001111111000000000

00000001111110000000000000000000

00000001111111000000000000000000

00000011111111110000000000000000

00000011111111111100000000000000

00000011111111111100000000000000

00000111111111111110000000000000

00000011111111111111000000000000

00000001111111111111110000000000

00000001111000001111110000000000

00000000000000000111110000000000

00000000000000000111110000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000011111000000000

00000000000000000111111000000000

00000000000000001111100000000000

00000000000000001111110000000000

00000000001111111111110000000000

00000000011111111111100000000000

00000000011111111111000000000000

Using this thought, we can reduce some ’extra parts’ of the number to reduce the computing time.