OCR of Hand-written Data using kNN

Goal

In this chapter:

- We will use our knowledge on kNN to build a basic OCR (Optical Character Recognition) application.
- We will try our application on Digits and Alphabets data that comes with OpenCV.

OCR of Hand-written Digits

Our goal is to build an application which can read handwritten digits. For this we need some training data and some test data. OpenCV comes with an image digits.png (in the folder opencv/samples/data/) which has 5000 handwritten digits (500 for each digit). Each digit is a 20x20 image. So our first step is to split this image into 5000 different digit images. Then for each digit (20x20 image), we flatten it into a single row with 400 pixels. That is our feature set, i.e. intensity values of all pixels. It is the simplest feature set we can create. We use the first 250 samples of each digit as training data, and the other 250 samples as test data. So let's prepare them first.

```
import numpy as np
import cv2 as cv

img = cv.imread('digits.png')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)

# Now we split the image to 5000 cells, each 20x20 size
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]

# Make it into a Numpy array: its size will be (50,100,20,20)
x = np.array(cells)

# Now we prepare the training data and test data
train = x[:,:50].reshape(-1,400).astype(np.float32) # Size = (2500,400)
test = x[:,50:100].reshape(-1,400).astype(np.float32) # Size = (2500,400)

# Create labels for train and test data
k = np.arange(10)
train_labels = np.repeat(k,250)[:,np.newaxis]
test_labels = train_labels.copy()

# Initiate kNN, train it on the training data, then test it with the test data with k=1
knn = cv.ml.KNearest_create()
knn.train(train, cv.ml.ROW_SAMPLE, train_labels)
ret,result,neighbours,dist = knn.findNearest(test,k=5)

# Now we check the accuracy of classification
# For that, compare the result with test_labels and check which are wrong
matches = result==test_labels
correct = np.count_nonzero(matches)
accuracy = correct*100.0/result.size
print( accuracy )
```

So our basic OCR app is ready. This particular example gave me an accuracy of 91%. One option to improve accuracy is to add more data for training, especially for the digits where we had more errors.

Instead of finding this training data every time I start the application, I better save it, so that the next time, I can directly read this data from a file and start classification. This can be done with the help of some Numpy functions like np.savetxt, np.savez, np.load, etc. Please check the NumPy docs for more details.

```
# Save the data
np.savez('knn_data.npz',train=train, train_labels=train_labels)

# Now load the data
with np.load('knn_data.npz') as data:
    print( data.files )
    train = data['train']
    train_labels = data['train_labels']
```

In my system, it takes around 4.4 MB of memory. Since we are using intensity values (uint8 data) as features, it would be better to convert the data to np.uint8 first and then save it. It takes only 1.1 MB in this case. Then while loading, you can convert back into float32.

OCR of the English Alphabet

Next we will do the same for the English alphabet, but there is a slight change in data and feature set. Here, instead of images, OpenCV comes with a data file, letter-recognition.data in opencv/samples/cpp/ folder. If you open it, you will see 20000 lines which may, on first sight, look like garbage. Actually, in each row, the first column is a letter which is our label. The next 16 numbers following it are the different features. These features are obtained from the UCI Machine Learning Repository. You can find the details of these features in this page.

There are 20000 samples available, so we take the first 10000 as training samples and the remaining 10000 as test samples. We should change the letters to ascii characters because we can't work with letters directly.

```
import cv2 as cv
import numpy as np

# Load the data and convert the letters to numbers
data= np.loadtxt('letter-recognition.data', dtype= 'float32', delimiter = ',',
                    converters= {0: lambda ch: ord(ch)-ord('A')})

# Split the dataset in two, with 10000 samples each for training and test sets
train, test = np.vsplit(data,2)

# Split trainData and testData into features and responses
responses, trainData = np.hsplit(train,[1])
labels, testData = np.hsplit(test,[1])

# Initiate the kNN, classify, measure accuracy
knn = cv.ml.KNearest_create()
knn.train(trainData, cv.ml.ROW_SAMPLE, responses)
ret, result, neighbours, dist = knn.findNearest(testData, k=5)

correct = np.count_nonzero(result == labels)
accuracy = correct*100.0/10000
print( accuracy )
```

It gives me an accuracy of 93.22%. Again, if you want to increase accuracy, you can iteratively add more data.

Additional Resources

1. Wikipedia article on Optical character recognition

Exercises

1. Here we used k=5. What happens if you try other values of k? Can you find a value that maximizes accuracy (minimizes the number of errors)?