Inf 2B Coursework-2
Task-1 Report
K-NN Classification

UUN: s1616497

April 13, 2018

Abstract

The aim of task one is to use K-NN classification in order to classify a dataset of images into one of 26 different classes: representing the letters of the English alphabet.

1 About dataset/ symbols used

1.1 Brief description

The task employs EMNIST handwritten character data set https://www.nist.gov/itl/ iad/image-group/emnist-dataset. Each character image is represented as 28-by-28 pixels in gray scale, being stored as a row vector of 784 elements ($28 \times 28 = 784$). A subset of the original EMNIST data set is considered in the task, restricting characters to English alphabet of 26 letters in either upper case or lower case. Each image's feature vector has 784 dimensions.

1.2 Symbols Used

1. Xtrn : The matrix of training data(Shape: 46800x784 = MxD)
2. Ctrn : The vector representing training data classes(Shape: 46800x1 = Mx1)
3. Xtst: The matrix of test data(Shape: 7800x784 = NxD)
4. Ks: The vector containing different K values for K-NN classification(Shape: Lx1)

2 Code Description

2.1 File 1: $my\_knn\_classify.py$

Parameter list:
$Xtrn$, $Ctrn$, $Xtst$, Ks

Return value:
$Cpreds$: NxL matrix of predicted classes of $Xtst$ matrix feature vectors

In order to proceed with the K-NN classification process, the distance matrix(DI) containing the distance from each test vector was calculated. To make this process quick and compact, instead of using nested for loops, the distance calculation was sped up by $Vectorization$.
$np.einsum()$ was used to calculate the XX and YY values as shown in Figure 2 as it is a bit faster than the method shown in Figure 1 below. $einsum()$ is used in such a way that the diagonal of the matrix product of $Xtst$, $Xtst.T$ and $Xtrn$, $Xtrn.T$ are set as the values of XX and YY respectively.
Functions used were: $np.einsum()$, $np.sum()$, $argsort()$, $np.dot$

Formula Used: DI = $(XX, ..., XX) - 2 * X * Y^T + (YY, ..., YY)^T$

Where:

- $X = Xtst$
- $Y = Xtrn$
- $XX = \begin{bmatrix} X_1 * X_1^T \\ \vdots \\ X_N * X_N^T \end{bmatrix}$
- $YY = \begin{bmatrix} Y_1 * Y_1^T \\ \vdots \\ Y_M * Y_M^T \end{bmatrix}$
- X_k, Y_k = the kth feature vector from $Xtst$ and $Xtrn$ respectively.

```
def calculateDistance(train, test):
    #Euclidean distance calculation on numpy arrays
    #distance = np.sum((train - test)**2, axis=-1)

    xx = np.sum(test**2, axis=1, keepdims=True, dtype=np.float16)
    yy = np.sum(train**2, axis=1, keepdims=True, dtype=np.float16)
    xy2 = -2*np.dot(test, train.T)

    DI = (xy2 + xx) + yy.T
    # Argsort sorts indices from closest to furthest neighbor, in ascending order
    sortDistance = np.zeros(np.shape(DI))
    sortDistance += DI.argsort()
    return DI, sortDistance
```

Figure 1: Snippet of distance calculation and sorting by indices.

```
def calculateDistance(train, test):
    #Euclidean distance calculation on numpy arrays
    xx = np.einsum('ij,ji->i', test, test.T, dtype=np.float)
    yy = np.einsum('ij,ji->i', train, train.T, dtype=np.float)

    xx = np.reshape(xx, (np.shape(test)[0], 1))
    xy2 = -2*np.dot(test, train.T)

    DI = (xy2 + xx) + yy
    # Argsort sorts indices from closest to furthest neighbor, in ascending order
    sortDistance = np.argsort(DI, axis=1)
    #sortDistance += DI.argsort()
    return sortDistance
```

Figure 2: Snippet of distance calculation using einsum and sorting by indices.

The datatypes of the elements in the matrices were changed to float 16 which also helped speed up the calculation procedure.

After obtaining the matrix with the sorted indices(sortDistance) as shown in Figure1, the matrix was sliced to obtain the first $maxk$ (i.e. maximum of Ks) columns as the these would be the only columns of interest to us in the classification.

The elements in the new sliced matrix(sortClasses) were then converted to their respective classes by using a vectorized lambda function. $lambda\ t : Ctrn[t]$

Finally, a vectorized stats.mode() function was used in a for loop iterating over the the different Ks to obtain the predicted classes of the vectors of $Xtst$.

2.2 File 2: *my_confusion.py*

Parameter list:

- *Ctrues* : The correct classes of the test vectors(Shape: Nx1)
- *Cpreds* : The predicted classes of the test vectors(Shape Nx1)

Return value:

- *CM* : K-by-K confusion matrix, where $CM[i][j]$ is the number of samples whose target is the $i'th$ class that was classified as j. K is the number of classes, which is 26 for the data set.
- *acc* : The ratio of right predictions to total predications

There was no need of vectorization in this case. Just a for-loop iterating over all the predications was used.

2.3 File 3: *my_knn_system.py*

The function *time.clock*() was used which is a part of the package *time*.
A list of file names was created. A $for - loop$, iterating L times was then used to generate, print out the desired outputs and write the confusion matrices to the respective .mat files.
Used *scipy.io.savemat*() to store the $Confusion\ Matrices$ in their respective files.

3 File 4: *printTable.py*

This file takes a dictionary of values and prints a table which is the desired output of the task.

4 Values Obtained

Accuracy	KValue	Prediction Size	Number of Errors
0.864230769231	1	7800	1059
0.869615384615	3	7800	1017
0.866666666667	5	7800	1040
0.859230769231	10	7800	1098
0.843333333333	20	7800	1222

For 3 nearest neighbors, the accuracy is the highest.
Time taken for *my_knn_classify.py*: 43.69 seconds