Digit Recognizer - MNIST

Siyuan Zhang

1 Resources

Student: Siyuan Zhang
Language: Python
Resource: MNIST data from Kaggle

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt, matplotlib.image as mpimg
from sklearn.model_selection import train_test_split
import time
import operator
```

2 Dataset details

There are two datasets, training data and testing data

1. The training data contains 28000 images
2. Each image has 784 pixels in total, each single pixel-value indicates the lightness or darkness of that pixel
3. The first column is the actual digit of the image
4. The testing data is the same with training data, but without "label" column to represent the actual number

Data Splitting

```
#read the data
dataset=pd.read_csv('train.csv')
images=dataset.values[0:,1:]
labels=dataset.values[0:,:1]
X_train, test_images, X_labels, test_labels=train_test_split(images,labels,random_state=2,test_size=0.2)
train_images,valid_images,train_labels,valid_labels =
train_test_split(X_train,X_labels,random_state=2,test_size=0.25)
```

First, I read the train.csv data called dataset, and I split the data into images and labels as two parts. Here, I split the dataset into three parts, training data (60%), validation data (20%) and testing data (20%). Firstly, I used the validation data to find the optimized K and test the testing data to check the accuracy.

Inspect the Dataset

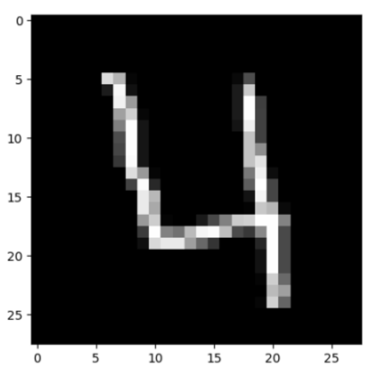
```
# print the training dataset
print(len(dataset))
print(dataset.head())
```

```
42000
   label  pixel0  pixel1  pixel2  ...  pixel780  pixel781  pixel782  pixel783
0      1       0       0       0  ...         0         0         0         0
1      0       0       0       0  ...         0         0         0         0
2      1       0       0       0  ...         0         0         0         0
3      4       0       0       0  ...         0         0         0         0
4      0       0       0       0  ...         0         0         0         0

[5 rows x 785 columns]
```

Dataset Visualization

We can plot an image in train dataset to visualize the data as a photo. For example, I chose the 4th row and reshape it into 28*28 matrix.



```
img = train_images.values[3].values
img = img.reshape(28, 28)
plt.imshow(img, cmap='gray')
plt.show()
```

3 Algorithm Description

1. Data pre-processing: After spiting the label and real data in the train.csv, I did binarization since the image pixel is from 0-255. If the value is not zero, then I modify it to one, which can speed up the processing time.

2. Distance metrics: which used to determine the distance between train data and test data.
   2.1. Subtract two matrixes and get a new matrix ‘m’
   2.2. Find the square of the matrix ‘m’ (that is, multiply the matrix itself)
   2.3. Add the values together by each line
   2.4. Perform a square root operation on the matrix ‘m’, then get the distance

```
# knn-algorithm
# input: current vector in test_images, train_images, train_labels, k
def classifyDigit(curr_image, dataSet, labels, k):
    # get how many rows in traindata
    dataSetSize = dataSet.shape[0]
    # get the distance
    diffMat = np.tile(curr_image, (dataSetSize, 1)) - dataSet
    sqDiffMat = diffMat ** 2
    sqDistance = sqDiffMat.sum(axis=1)
    distances = sqDistance ** 0.5
    # sort by distance
    sortedDistIndicies = distances.argsort()
    # get kth shortest distance images
    classCount = {}
    for i in range(k):
        curr_image = labels[sortedDistIndicies[i]]
        classCount[curr_image] = classCount.get(curr_image, 0) + 1
    # traverse to find the item appears most
    maxCount = 0
    answer = ""
    for k, v in classCount.items():
        if v > maxCount:
            maxCount = v
```

```
        answer = k
    return answer
```

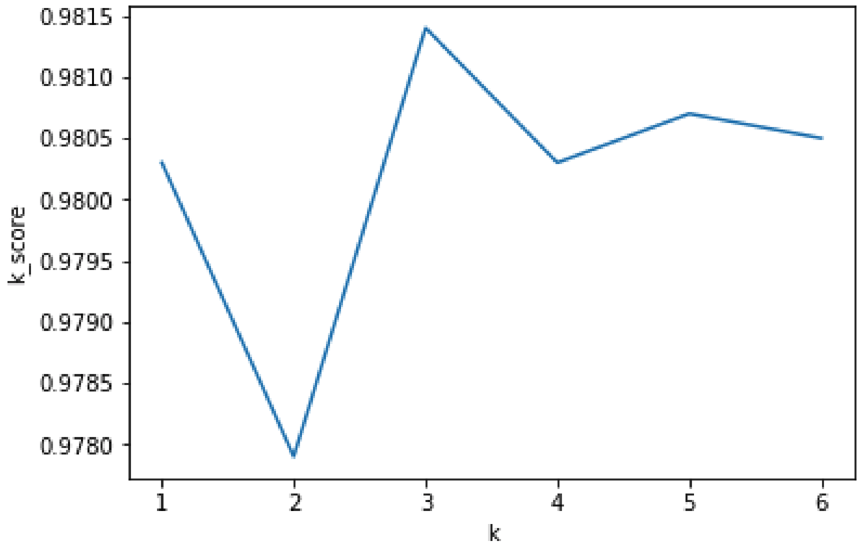
4 Algorithm Results

Selection of K

Different selection of K will affect the accuracy significantly. To save time, here I selected ‘test_images’ mentioned above as sample to check the different accuracy with different K, by using the knn algorithm described above.

```
def diffK():
    k_score = []
    for k in range(1, 6 + 1):
        print("k = {} Training.".format(k))
        start = time.clock()
        score = check(valid_images, valid_labels, k)
        end = time.clock()
        k_score.append(score)
        print("Score: {}.".format(score))
        print("Complete time: {} Secs.".format(end - start))
    print(k_score)
    plt.plot(range(1, 6 + 1), k_score)
    plt.xlabel('k')
    plt.ylabel('k_score')
    plt.show()
```

```
[0.9803, 0.9779, 0.9814, 0.9803, 0.9807, 0.9805]
```



From the diagram we can know clearly that when k = 3, the accuracy is the best. So I simply chose k = 3 to predict the test dataset.

Algorithm accuracy

Use the testing data to check the accuracy.

```
m = test_labels.shape[0]
resultList = []
errorNum = 0
for i in range(m):
    curResult = classify(test_images[i], train_images, train_labels, 3)
    resultList.append(int(curResult))
    if (int(curResult) != test_labels[i]):
        errorNum += 1.0
    print("\nthe total number of errors is: %d" % errorNum)
    print(i)
    print("#########################################")
print("\nthe total error rate is: %f" % (errorNum / float(m)))
```

```
the total error rate is: 0.037381
```

The accuracy is 96.26%.

Confusion Matrix

```
cm = confusion_matrix(test_labels, resultList)
print(cm)
```

```
[[814   2   1   0   0   1   2   1   0   0]
 [  0 953   2   1   0   0   0   3   2   1]
 [  2   6 787   7   0   2   4  18   1   2]
 [  0   1   3 833   0  12   1   6   6   2]
 [  1   8   0   0 813   0   3   0   0  31]
 [  3   0   0  16   1 691   8   0   4   6]
 [  8   1   0   0   1   2 827   0   0   0]
 [  0  10   4   0   4   0   0 843   0  12]
 [  2   5   2  12   1  19   4   3 733  12]
 [  4   1   1   5  13   1   1  16   0 792]]
```

The confusion matrix is shown above, and we can know that the number of correct answers is 8086, the total number of images is 8400. So, the accuracy is 96.26%.

Check Single K

I select the 4th number in test set to check the variance by different K, where the actual number is 9.

```
#k = 1
result = int(classifyDigit(test_images[3], train_images, train_labels, 1))
print (result)
```

[0]

```
#k = 2
result = int(classifyDigit(test_images[3], train_images, train_labels, 2))
print (result)
```

[0]

```
#k = 3
result = int(classifyDigit(test_images[3], train_images, train_labels, 3))
print (result)
```

[9]

```
#k = 4
result = int(classifyDigit(test_images[3], train_images, train_labels, 4))
print (result)
```

[9]

```
#k = 7
result = int(classifyDigit(test_images[3], train_images, train_labels, 7))
print (result)
```

[9]

From the result, it turns out that when k is too small, like 1 or 2, the accuracy is not good. When the value of k is larger than 2, the accuracy is fine.

5 Runtime

For d dimension, we need $O(d)$to compute one distance between two data, then we need to sort the distance, which takes $O(nlogn)$, at last, we need to select k nearest neighbors. In total, the runtime is $O(n(d + nlogn + k))$ to classify the data.

I have spitted the data into 3 parts, the last 20% is the testing data, which includes 8400 images:

```
start = time.clock()
m = test_labels.shape[0]
resultList = []
errorNum = 0
for i in range(m):
    curResult = classify(test_images[i], train_images, train_labels, 3)
    resultList.append(int(curResult))
    if (int(curResult) != test_labels[i]):
        errorNum += 1.0
end = time.clock()
print ('Time used: {}'.format(end - start))
```

```
Time used: 2570.238005
```