CSC 535 Data Mining

Assignment 2 Report Collection

Submitted to:

Dr. Jamil Saquer

Author:

*Caleb Sutton*

**Report**

**Introduction**

*For this assignment students were required to use a portion of the MNIST data set. The MNIST data set is a collection of greyscale pixel values which correspond to each pixel of a 28px x 28px image. Each of these images are of a hand drawn numerical digit. For our assignment the data is organized into two .csv files, both of which have one image per line. In other words, there are 784 comma separated pixel values on each line, plus the correct classification at the beginning of each line. The assignment was for students to use the MNIST\_train file to classify each of the data samples in the MNIST\_test file using a KNN algorithm.*

**Background**

*The KNN algorithm works by comparing a test sample to each of the samples in a training set and determining the ‘K’ closest samples. ‘K’ can be any number as long as it is not larger than the training set, and it has a large impact on the accuracy of the KNN algorithm. In a simple KNN algorithm the most common class in the K closest samples is the class chosen for the test sample. For this assignment, a weighted voting method was used, where each of the K closest samples only counts as a fraction of a vote depending its calculated distance.*

*To determine the distance between two samples you first need to know which distance measure you will be implementing. For this assignment it was suggested that we use Euclidean distance. Euclidean distance is basically and extension of Pythagorean theorem. It can be calculated by taking the squaring the difference of every field between two samples, summing those squares, and then taking the square root of that sum. For this assignment Euclidean distance was a straightforward distance measure.*

**Implementation**

*My implementation of this algorithm was fairly simple. I take the training data file, test data file, and value of ‘K’ as command line arguments. I then, read in the csv files using python’s csv library, the readCSV function in my program organizes the files into lists of lists. From there I iterate through the test data file, calling my KNN classify function on each of the samples. The function returns if the classification was successful or not, which I track in order to report performance statistics.*

*Inside my KNN classification function, which is called classifySample() in my program I iterate through all of the training samples and compute the distance between each training sample and the test sample. This is done using the calculateDistance() function which returns the Euclidean distance between two samples. After the distance of a sample is calculated, it is inserted into the correct position of a sorted list of length K, and then the sample with the greatest distance is popped from that list. After every training sample has been considered the classification for the test sample is computed and compared to the actual value.*

**Experimental Setup and Results**

*To test this algorithm, I experimented with the value of K and measured the accuracy. I found that a value of 7 or 9 for K is the best for this algorithm, and particular data set. The highest accuracy I achieved was 90% with K being 7 or 9. I think it is worth mentioning that the accuracy seems to hit floor of 82% as the value of K continues to increase. To mean this analogous to big O notation in programming. The worst-case accuracy performance in this case seems to be 82%*

**Bonus**

*I modified my original KNN algorithm to run across multiple cores in an attempt improve the runtime performance of the KNN algorithm. While, this did not have a direct impact on the accuracy performance of my KNN algorithm I was able to reach 96% accuracy by using a larger training data set. The larger training data set called train.csv in my hw2 folder, which I got from* <https://www.kaggle.com/c/digit-recognizer>*, contains 42,000 samples, as opposed to the 949 in MNIST\_train. My computer at home has an 8 core processor so I was really able to take advantage of this improvement. The larger data set took between 280-300 seconds to run on the parallelized KNN and judging from my smaller tests it would have taken around 5 times longer or close to 25 minutes to run on the unparallelized KNN.*

*One important thing I would like to mention in my testing of the parallelized KNN on this data set is that I had to alter program to ignore exact matches. I’m pretty sure the MNIST\_test samples are a subset of the train.csv samples. The algorithm has 100% accuracy when you allow exact matches, which I did not think was fair in my testing.*

*This is my code in hw2\_parallel.py where I attempt to disregard exact matches.*

if result['distance'] == 0:

result['distance'] = 999999

else:

result['vote'] = 1/result['distance']

**Conclusion**

*In general, I am pleased with my current algorithm. I would have liked for it to be more accurate, but most of the MNIST examples I found were using neural nets to achieve greater accuracy. I think I would consider my parallelized KNN as more of a luxury than a true performance improvement. It was really nice for testing values of K, because it only took around 6 seconds to run as opposed to 30 seconds, but you could achieve the same accuracy using the normal KNN as long as you had more time.*

**Code (hw2.py)**

"""

Program: hw2.py

Programmed By: Caleb Sutton

Description: Implementation of KNN algorithm on MNIST

Trace Folder: Sutton728/hw2

"""

# imports

import sys # used for csv input

import csv # used for csv input

import math # used for sqrt() in calculateDistance()

import time # used to time algorithm

# program main

def main():

# Check to ensure program is being used properly

if (len(sys.argv) != 4):

print("\nUsage: python hw2.py <trainingDataFile> <testDataFile> <k>\n")

exit()

# Start a timer for measuring program speed/performance

startTime = time.time()

# Variables used in main

trainingData = []

testData = []

k = 0

numCorrect = 0

numIncorrect = 0

accuracyRate = 0.0

# Initialize training data, test data, and K

trainingData = readCSV(sys.argv[1])

testData = readCSV(sys.argv[2])

k = int(sys.argv[3])

# Print k

print('\nK = ' + str(k) + '\n')

# For each test sample in the testing data call the classify

# sample function, keeping track of the number if correct

# and incorrect samples

for testSample in testData[1:]:

if classifySample(trainingData[1:], testSample, k) == True:

numCorrect += 1

else:

numIncorrect += 1

# Compute the accuracy

accuracyRate = numCorrect / (numCorrect + numIncorrect)

round(accuracyRate, 4)

# Print results

print('\nAccuracy Rate: ' + str(accuracyRate \* 100) + '%')

print('Number of misclassified test samples: ' + str(numIncorrect))

print('Total number of test samples: ' + str(numCorrect + numIncorrect) + '\n')

# Print time elapsed

endTime = time.time()

print('Time Elapsed: ' + str(round((endTime-startTime), 3)) + 's\n')

# classifySample() function takes a set of training data, test

# sample, and k value as parameters and returns whether or

# not the test samples computed class mathces its actual

# class

def classifySample(trainingData, testSample, k):

# list for holding the nearest neighbors to the test sample

nearestSamples = []

# loop for each sample in the training data and calculate

# its distance from the test sample and maintain a sorted

# list of length k of the nearest neighbors

for trainingSample in trainingData:

# result is a dictionary used for keeping track of

# the nearest neighbors

result = {}

result['class'] = trainingSample[0]

result['distance'] = calculateDistance(testSample[1:], trainingSample[1:])

if result['distance'] == 0:

result['vote'] = 1

else:

result['vote'] = 1/result['distance']

# if there are less than k samples in the list append

# results and then sort the array

if len(nearestSamples) < k:

nearestSamples.append(result)

nearestSamples.sort(key = lambda sample: sample['distance'], reverse = False)

# we only get to this else once the list contains k

# samples, which means we can iterate through and insert

# the new sample in the correct postion, if its distance

# is larger than all the samples it will be immediately

# popped, otherwise the greatest distance will be popped

else:

i = 0

for sample in nearestSamples:

if result['distance'] < sample['distance']:

nearestSamples.insert(i, result)

nearestSamples.pop()

break

i += 1

# call organizational function calculateClass() to calculate

# the class of the test sample using the nearestSamples list

computedClass = calculateClass(nearestSamples)

print('Desired Class: ' + str(testSample[0]) + ', Computed Class: ' + str(computedClass))

# return true if the calculated class mathces the actual class

# otherwise return false

if int(computedClass) == int(testSample[0]):

return True

else:

return False

# calculateDistance() takes two samples as parameters and returns

# the calculated euclidean distance bewteen the two

def calculateDistance(sample1, sample2):

distance = 0

for i in range(len(sample1)):

square = int(sample1[i]) - int(sample2[i])

distance += square \* square

return math.sqrt(distance)

# calculateClass() is a helper function to classifySample() it

# takes the nearestSamples list and then computes which class

# had the highest number of votes and returns it

def calculateClass(nearestSamples):

votes = [0,0,0,0,0,0,0,0,0,0]

highest = 0

highestValue = 0

for sample in nearestSamples:

votes[int(sample['class'])] += sample['vote']

for i in range(len(votes)):

if votes[i] > highestValue:

highestValue = votes[i]

highest = i

return highest

# readCSV() takes a file path to a .csv file as a parameter

# and returns a list of lists representing the csv file

def readCSV(filepath):

data = []

with open(filepath) as csvfile:

readCSV = csv.reader(csvfile, delimiter = ',')

for row in readCSV:

data.append(row)

return data

main()

**Code (hw2\_parallel.py)**

*Hw2\_prallel.py can be found in my trace folder (sutton728/hw2). It is largely the same as hw2.py with the addition of this code in main() and an extra loop in classifySamples():*

# calculate the number of samples to be classified by

# each core

for i in range(cpuCount):

samplesPerCore[i] = len(testData[1:]) / cpuCount

if i < len(testData[1:]) % cpuCount:

samplesPerCore[i] += 1

# start each process using the classifySamples() function

# giving it the training data, and a chunk of the test samples

startIndex = 1

for i in range(cpuCount):

samples = testData[int(startIndex) : int(startIndex + samplesPerCore[i])]

p = mp.Process(target = classifySamples, args = [trainingData[1:], samples, k, que], name = 'classify\_process\_' + str(i))

processes.append(p)

p.start()

startIndex += samplesPerCore[i]

# wait for processes to finish

for p in processes:

p.join()

# get the results from each process using the que

for i in range(que.qsize()):

result = que.get()

if result == True:

numCorrect += 1

else:

numIncorrect += 1