CSC 535 Data Mining

## Assignment 3 Report

### Submitted to:

### Dr. Jamil Saquer

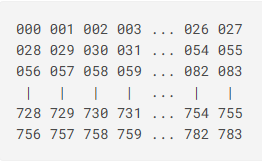
### Author(s):

### David Gray

**Application of KNN algorithm on classification datasets**

**Introduction**

This report is about a digit recognizer software that classifies objects using the K Nearest Neighbors (KNN) algorithm to determine what single digit is written. This computer vision task is useful in real life situations such as creating a text document from a pdf file. The dataset used to train and test the software is comprised of over 900 objects that each contain 784 pixels. Each pixel contains an integer value between 0 and 255, inclusive and is arranged in a twenty-eight by twenty-eight array forming an image of the drawn digit, shown by Figure 1.



*Figure 1: Arrangement of pixels*

*Source: Digit Recognizer*

**Background**

The KNN algorithm takes in the training data, testing data, and number of nearest neighbors and outputs the predicted classification based on the classes of the nearest neighbors. Since KNN does not build a model classification is expensive. This is because for every object the algorithm classifies the distances between the object being classified and all the objects in the training data are calculated and compared. This software uses the Euclidean distance formula to calculate distance, which works well with the high dimensional dataset being used. To further improve the accuracy, I compare the nearest neighbors, found using the Euclidean distance formula, to each other and add a weight to each neighbor, based on the distance to the unclassified object, that affects the worth of its vote on classification of the unclassified object.

**Implementation**

Part of this assignment was to add a method to account for weighted voting. Initially, I converted distances into percentages based on the furthest neighbor. This proved to be counter intuitive and after further research I instead used the inverse function to weight distances. The solution makes my code simpler, saves computing time, and produces more accurate results. In the code below, I have commented out the method for comparing by percentages.

def inverseFunction(x):

    return 1/x

def weightedVote(neighbors, t):

#     distances = np.array([neighbor['distance'] for neighbor in neighbors])

#     div = distances.max()

    for neighbor in neighbors:

        # Compare using percents

#         neighbor['weight'] = abs((div - neighbor['distance'])/div)+1

        # Compare distance with 1/x

        neighbor['weight'] = inverseFunction(neighbor['distance'])

    cls\_and\_distance = pd.DataFrame([{'class':neighbor['obj'].loc['label'],'weight':neighbor['weight']} for neighbor in neighbors])

    weight\_sums = cls\_and\_distance['weight'].groupby(cls\_and\_distance['class']).sum()

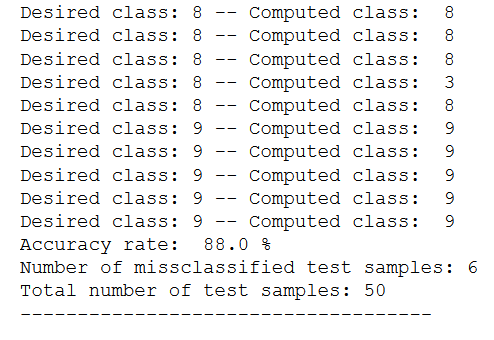
    max\_vote = weight\_sums.max()

    class\_vote = weight\_sums[weight\_sums == max\_vote].keys()[0]

    return class\_vote

**Experimental Setup and Results**

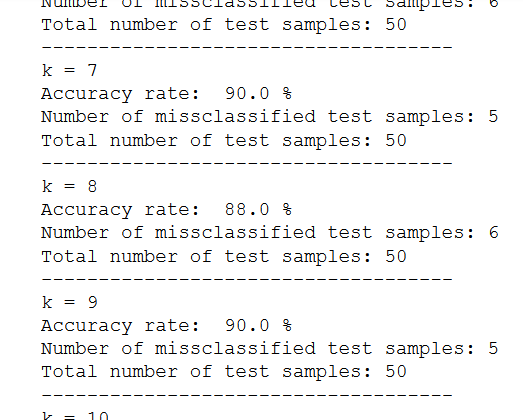
I created my KNN algorithm to use a standard inverse function to compute weights from distances. Using the five nearest neighbors the algorithm achieved an accuracy rate of eighty-eight percent, misclassifying six of fifty samples as shown in Figure 2. The k-value was chosen by trial and error taking about twenty minutes for each k-value.



*Figure 2: Results of KNN using inverse function (k=5)*

**Conclusion**

Choosing the k-value via trial and error was not an optimal strategy because KNN is a lazy learner and each classification of the testing dataset took approximately twenty minutes. I solved this problem by storing an array containing each unclassified object, the distances in relation to each object in the training dataset, and the classification of each object in the training dataset. Using this method, I reduce the number of times the KNN algorithm calculates distances for each unclassified object to one. This allows me to loop through a range of k-values and quickly receive results in a fraction of the time. For example, testing a number of k-values, n, using the trial and error method requires n\*(classification time), but using my solution decreases n to equal 1. After processing a large range of k-values the software determined that the values seven and nine gave the highest accuracy at ninety percent, as seen in Figure 3. The solution improved the accuracy of my trial and error method by two percent, but the real benefit is shortening the total time spent finding the optimal k-value.



*Figure 3: Optimized k-value results*

**References**

Digit Recognizer. (n.d.). Retrieved October 20, 2020, from https://www.kaggle.com/c/digit-recognizer/data?select=sample\_submission.csv

**Code**

"""

Program: hw3.ipynb

Author: David Gray

Description: Homework 3: A KNN algorithm that uses Euclidean distance and weighted voting via an inverse function of distance to classify objects.

Extra Credit: Functions available for finding the optimal k-value efficiently.

"""

import pandas as pd

import numpy as np

import math

def inverseFunction(x):

    # Returns the inverse of x

    return 1/x

def distance(p, q):

    # Returns the Euclidean distance between two points

    distance = 0

    for i in range(1,len(p)):

        distance += (p[i]-q[i])\*\*2

    return math.sqrt(distance)

def getNeighbors(D, k, t):

    # Returns the k nearest neighbors

    neighbors = []

    total = len(D)

    for index in range(total):

        d = D.loc[index]

        dist = distance(t,d)

        if len(neighbors) < k:

            neighbors.append({'obj':d, 'distance':dist})

        elif [dist < neighbor['distance'] for neighbor in neighbors]:

            # Replace the neighbor that has the greatest distance from t with d

            distance\_array = np.array([n['distance'] for n in neighbors])

            farthest = distance\_array.max()

            for i in range(len(neighbors)):

                if distance\_array[i] == farthest:

                    far\_n = i

            neighbors[far\_n] = {'obj':d, 'distance':dist}

    return neighbors

def weightedVote(neighbors, t):

    # Returns the predicted classification based on weighted votes of nearests neighbors

#     distances = np.array([neighbor['distance'] for neighbor in neighbors])

#     div = distances.max()

    for neighbor in neighbors:

        # Compare using percents

#         neighbor['weight'] = abs((div - neighbor['distance'])/div)+1

        # Compare distance with 1/x

        neighbor['weight'] = inverseFunction(neighbor['distance'])

    cls\_and\_distance = pd.DataFrame([{'class':neighbor['obj'].loc['label'],'weight':neighbor['weight']} for neighbor in neighbors])

    weight\_sums = cls\_and\_distance['weight'].groupby(cls\_and\_distance['class']).sum()

    max\_vote = weight\_sums.max()

    class\_vote = weight\_sums[weight\_sums == max\_vote].keys()[0]

    return class\_vote

def KNN(D, k, t):

    # KNN algorithm

    neighbors = getNeighbors(D, k, t)

    predicted\_class = weightedVote(neighbors, t)

    return predicted\_class

def basicClassification():

    # Classification using KNN

    train\_dataset = pd.read\_csv("MNIST\_train.csv")

    test\_dataset = pd.read\_csv("MNIST\_test.csv")

    k=5

    sample\_count = 0

    correct\_count = 0

    print("k =", k)

    for index in range(len(test\_dataset)):

        obj = test\_dataset.loc[index]

        actual\_class = obj.loc['label']

        predicted\_class = KNN(train\_dataset, k, obj)

        print("Desired class:",actual\_class, "-- Computed class: ", predicted\_class)

        sample\_count += 1

        if actual\_class == predicted\_class:

            correct\_count += 1

    accuracy = (correct\_count/sample\_count)\*100

    print("Accuracy rate: ", accuracy, "%")

    print("Number of missclassified test samples:", sample\_count-correct\_count)

    print("Total number of test samples:", sample\_count)

    print("------------------------------------")

def predictClass(obj\_model, k):

    # Returns the predicted classification based on weighted votes of nearests neighbors

    neighbors = obj\_model[:k]

    for n in neighbors:

        n['weight'] = inverseFunction(n['distance'])

    cls\_and\_distance = pd.DataFrame([{'class':neighbor['obj'].loc['label'],'weight':neighbor['weight']} for neighbor in neighbors])

    weight\_sums = cls\_and\_distance['weight'].groupby(cls\_and\_distance['class']).sum()

    max\_vote = weight\_sums.max()

    class\_vote = weight\_sums[weight\_sums == max\_vote].keys()[0]

    return class\_vote

def modifiedKNN(model, k):

    # Classification using KNN where distances have already been calculated

    sample\_count = 0

    correct\_count = 0

    print("k =", k)

    for index in range(len(model)):

        obj\_model = model[index]

        obj = obj\_model[0]['obj']

        actual\_class = obj['label']

        predicted\_class = predictClass(obj\_model[1:], k)

        sample\_count += 1

        if actual\_class == predicted\_class:

            correct\_count += 1

    accuracy = (correct\_count/sample\_count)\*100

    print("Accuracy rate: ", accuracy, "%")

    print("Number of missclassified test samples:", sample\_count-correct\_count)

    print("Total number of test samples:", sample\_count)

    print("------------------------------------")

def sortArray(array):

    # Returns a sorted array

    for i in range(2,len(array[1:])):

        item\_to\_insert = array[i]

        j = i - 1

        while j >= 1 and array[j]['distance'] > item\_to\_insert['distance']:

            array[j + 1] = array[j]

            j -= 1

        array[j + 1] = item\_to\_insert

    return array

def build\_model(train\_D, test\_D):

    # Calculates and stores distances based on KNN

    complete\_model = []

    total = len(train\_D)

    for index\_t in range(len(test\_D)):

        t = test\_D.loc[index\_t]

        model\_per\_obj = []

        model\_per\_obj.append({'obj':t})

        for index\_d in range(total):

            d = train\_D.loc[index\_d]

            dist = distance(t,d)

            model\_per\_obj.append({'obj':d, 'distance':dist})

        sort\_model\_per\_obj = sortArray(model\_per\_obj)

        complete\_model.append(sort\_model\_per\_obj)

    return complete\_model

def optimizedKClassification():

    # Build model

    train\_dataset = pd.read\_csv("MNIST\_train.csv")

    test\_dataset = pd.read\_csv("MNIST\_test.csv")

    model = build\_model(train\_dataset, test\_dataset)

    # Find optimized k-value

    k\_range = round(math.sqrt(len(train\_dataset)))

    if k\_range%2 == 0:

        k\_range-=1

    for K in range(1,k\_range):

        modifiedKNN(model, K)

basicClassification()

optimizedKClassification()