# Purpose

This project was a practice implementation of a Naïve-Bayes classifier. As it was conducted for didactic purposes, the implementation was performed without the use of existing machine learning libraries.

# Description of Data

The data being classified is Modified National Institute of Standards and Technology (MNIST) handwriting examples for numeric characters. Each observation is a 28x28 image stored as a single array 28^2 in length. Each value corresponds to the color of the pixel: 0 being white and 255 being black.

The subset utilized for this project contained 30,0003 training examples and 10,000 test examples.

# Methods

Data was imported and transformed such that 0 remained 0 and any other value became 1. This allows the model to be simplified while still retaining visibly distinguishable data, as shown below.

![A screenshot of a cell phone

Description automatically generated]()

Two classes where created; one for individual pixels and one for the classifier. The Pixel class contained two arrays of importance: the probability for each possible outcome given that the observed pixel value is 0, and the probability for each possible outcome given that the observed pixel value is 1. These arrays were created by an update function within the class that accepts two arrays as inputs: all values for the pixel within the data, and the true classes for the training data. From this, the probability of the pixel being 0 or 1 given each true class was calculated and stored. This class also stores the probability of the pixel being 1, from which the probability of it being 0 is simply (1-p (1)).

*Probability Pixel == 0: [ P(0|0), P(1|0), … P(9|0) ]*

*Probability Pixel == 1: [ P(0|1), P(1|1), … P(9|1) ]*

The second class, the classifier, instantiates by reading the length of the dataset’s first feature and instantiating an appropriate number of pixels objects. It then passes to each pixel object the respective evidence and class arrays to calculate the probabilities of each pixel given each outcome. The classifier object also contained a function to classify new inputs. The classification function iterates through each pixel and constructs a list of the probabilities for each possible class by using the Bayes Theorem:

The second function was used in a loop that iterated through the test dataset and stored counts of correct classifications and total classifications. These counts were then used to construct confidence intervals using:

Where

* z = 1.96 for 95% confidence
* n = the count of tests within the class
* error = (1-(correct classifications / total trials within the class))

These values were then added and subtracted from the observed accuracy rates for each digit.

# Results

﻿Out of 10000 tests, 8400 were correct, a ratio of 0.84.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Total | Correct | Ratio | 95% Confidence Interval |
| 0 | 980 | 884 | 0.902041 | 0.883429-to-0.920652 |
| 1 | 1135 | 1080 | 0.951542 | 0.939049-to-0.964035 |
| 2 | 1032 | 852 | 0.825581 | 0.802429-to-0.848734 |
| 3 | 1010 | 841 | 0.832673 | 0.809653-to-0.855694 |
| 4 | 982 | 795 | 0.809572 | 0.785014-to-0.834130 |
| 5 | 892 | 635 | 0.711883 | 0.682162-to-0.741604 |
| 6 | 958 | 843 | 0.879958 | 0.859377-to-0.900539 |
| 7 | 1028 | 866 | 0.842412 | 0.820139-to-0.864686 |
| 8 | 974 | 756 | 0.776181 | 0.750004-to-0.802357 |
| 9 | 1009 | 848 | 0.840436 | 0.817840-to-0.863032 |

The same results visualized:

![A screenshot of a cell phone

Description automatically generated]()

The results are somewhat surprising, as I did not expect 5 to be the least accurately classified digit. However, across numerous iterations of segregating the data and retesting, this remained relatively consistent. I hypothesize that this may be because 5 tends to be the least consistently written digit, and the lack of consistency within the data tends to be expressed in the model by making it less likely to generalize well to future observations.

# Appendix

The libraries used to run this classifier are:

* csv – for data import
* random – for reordering data randomly
* copy – used in re-ordering data to be non-destructive with respect to the original order
* math – for an optimized square root function
* numpy – used for containers of data within the image generator function
* PIL – used to actually construct the image file
* Matplotlib – used for the bar plot shown above

﻿

All of these libraries are installed alongside the IDE used for development, Spyder, which is itself a component installed with the Anaconda package manager. It can be downloaded here: <https://www.anaconda.com/distribution/>

With Anaconda installed, placing the MNIST files and the Python script itself in the same directory, then opening the Python script in Spyder, is sufficient preparation to run the script.

The code itself:

﻿#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created on Fri Oct 11 17:53:47 2019

@author: jmcleod

"""

import csv,random,copy,math

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

#from importlib import reload # not needed for python 2

#reload(plt)

#%%

def data\_import(file):

'''

This function imports the data from a particular file

and returns an array of arrays

NOTE: I am normalizing pixel values to simply be 0 and 1: no in between.

'''

data = []

with open(file, 'r') as csvfile:

csv\_r = csv.reader(csvfile,delimiter=' ')

for row in csv\_r:

row\_nums = []

for i in range(len(row)):

try:

val = float(row[i])

if i > 0:

val = round(val/255,4)

if val > 0: # making inputs binary for simplicity

val =1

# The above line scales the data imported

row\_nums.append(val)

except:

print('ERROR on import: non-numerical data')

print(row[i])

break

data.append(row\_nums)

return(data)

def data\_import\_loop(string,denom):

'''

This function loops the data import across all files of the chosen type,

which is specified by the string argument passed to the function.

It then uses the first value in the set to add the imported arrays

to the correct dictionary key, created with values 0-9.

The resulting dictionary is returned.

'''

files = []

data\_dict = {}

for i in range(10):

file\_name = string+str(i)+'.txt'

files.append(file\_name)

data\_dict[i]=[]

for i in files:

data = data\_import(i)

for j in range(len(data)):

if j%denom==0: # SUBSET data to 1/5th

data\_dict[data[j][0]].append(data[j][1:])

return(data\_dict)

def create\_image\_data(char\_matrix):

'''

This function outputs a human-viewable copy of an input from matrix form

'''

data = np.zeros( (len(char\_matrix),len(char\_matrix[0]),3), dtype=np.uint8 )

for row in range(len(char\_matrix)):

for col in range(len(char\_matrix[row])):

#print(col,row,len(char\_matrix),len(char\_matrix[0]))

val = 255 - char\_matrix[col][row]

data[row,col] = [val,val,val]

return(data)

def create\_large\_image(data\_dict):

'''

This function just creates an image from the data using 10 of each

input variable and all ten inputs for a 10x10 image instead of a single

input

'''

shortest = 1000000

for k,v in data\_dict.items():

if len(v) < shortest:

shortest = len(v)

big\_matrix\_data = []

for m in range(10):

medium\_matrix\_data = []

for i in range(28):

medium\_matrix\_data.append([])

for i in range(10):

random\_num = random.randint(0,shortest-1)

array = data\_dict[m][random\_num]

for j in range(len(array)):

medium\_matrix\_data[j%28].append((array[j]\*255))

for i in medium\_matrix\_data:

big\_matrix\_data.append(i)

big\_image = create\_image\_data(big\_matrix\_data)

image = Image.fromarray(big\_image)

image.show()

def randomize\_data\_arrays(data\_dict):

'''

This function is a hold-over from the NN I created; it needed randomized

data. While the Naive Bayes classifier doesn't need the data to be

randomized, I kept this so that my sets would be easier to spot check

a number of different features easily. In practice, I would remove this

to reduce the computational expense of the algorithm.

'''

data\_array = []

data\_result = []

for k,v in data\_dict.items():

for i in v:

data\_result.append(k)

data\_array.append(i)

random\_index = []

for i in range(len(data\_array)):

random\_index.append(random.random())

random\_index\_copy = copy.deepcopy(random\_index)

rand\_data\_array = []

rand\_data\_result = []

for i in range(len(random\_index)):

min\_val = min(random\_index\_copy)

random\_index\_copy.pop(random\_index\_copy.index(min\_val))

index\_val = random\_index.index(min\_val)

rand\_data\_array.append(data\_array[index\_val])

rand\_data\_result.append(data\_result[index\_val])

data\_array = rand\_data\_array

data\_result = rand\_data\_result

return(data\_array,data\_result)

def reshape\_data(data):

'''

This function rotates a matrix 90 degrees.

[[x,x],[y,y]] -> [[x,y],[x,y]]

'''

data\_reshaped = []

for i in range(len(data[0])):

data\_reshaped.append([])

for i in range(len(data)):

for j in range(len(data[i])):

data\_reshaped[j].append(data[i][j])

return(data\_reshaped)

class pixel:

'''

This class is used to learn and store the probabilities for each input

attribute.

'''

def \_\_init\_\_(self):

self.outcomes = [] # just a list of the possible classes

self.pixel\_proba = [] # probability of the evidence

self.outcome\_proba = [] # posterior probability

self.outcome\_proba\_neg = [] # posterior probability if evidence is (1-p)

for i in range(10):

self.outcomes.append(i)

self.outcome\_proba.append(0)

self.outcome\_proba\_neg.append(0)

def update\_probas(self,array,target):

pos\_count\_dict,neg\_count\_dict = {},{}

count\_0,count\_1 = 0,0

'''Adds up counts to use to calculate probabilities of the evidence

and posterior probabilities'''

for i in self.outcomes:

pos\_count\_dict[i],neg\_count\_dict[i]=0,0

for i in range(len(array)):

if array[i] == 0:

neg\_count\_dict[target[i]]+=1

count\_0+=1

else:

pos\_count\_dict[target[i]]+=1

count\_1+=1

self.pixel\_proba = count\_1 / (count\_1+count\_0)

'''

Calculate the outcome likelihoods given a positive or negative value

for this pixel. As in, what is the probability the actual image is

a 1 given this pixel being 0 or 1, and what is the probability it is

a 2 given this pixel being 0 or 1. '''

for k,v in pos\_count\_dict.items():

try:

self.outcome\_proba[k] = v / (v+neg\_count\_dict[k])

except:

print("Error: more outcomes allowed than are present in data")

for k,v in neg\_count\_dict.items():

try:

self.outcome\_proba\_neg[k] = v / (v+pos\_count\_dict[k])

except:

print("Error: more outcomes allowed than are present in data")

class naive\_bayes:

'''

This is the setup of the classifieer. It creates a set of objects

representing the inputs and then learns the likelihood of the outputs

for each

'''

def \_\_init\_\_(self,data,target):

self.pixels = {}

self.target\_proba = [1]\*10

for i in range(len(data)):

self.pixels[i]=pixel()

self.pixels[i].update\_probas(data[i],target)

temp = {}

for i in target:

if i not in temp:

temp[i]=1

else:

temp[i]+=1

for i in range(len(self.target\_proba)):

self.target\_proba[i]=temp[i]/len(target)

'''

This function makes classifications given a new input array by

Calculating the likelihood of the evidence given the outcome for each

pixel, divided by the likelihood of the evidence.

The output is an arrray of probabilities for each digit. The highest of

these is the class returned by the model.

'''

def classify(self,array):

posteriors,evidence,results = [1]\*10, [1]\*10, []

for i in range(len(array)):

p\_e = self.pixels[i].pixel\_proba

if array[i] == 1:

p\_p = self.pixels[i].outcome\_proba

else:

p\_p = self.pixels[i].outcome\_proba\_neg

p\_e = 1-p\_e

for j in range(len(posteriors)):

posteriors[j] = posteriors[j] \* p\_p[j]

evidence[j] = evidence[j] \* p\_e

for i in range(len(posteriors)):

try:

results.append((posteriors[i]\*self.target\_proba[i])/evidence[i])

except:

results.append(0)

summation = 0

for i in results:

summation+=i

if summation == 0:

summation =1

for i in range(len(results)):

results[i] = results[i]/summation

return(results)

#%%

'''

Data Import

Denom is a variable used to determine how much of the data to import: 1/denom

The string values 'train' and 'test' are for the filenames.

'''

denom = 2

data\_dict = data\_import\_loop('train',denom)

denom = 1

test\_dict = data\_import\_loop('test',denom)

'''

Check Data Import

This prints the counts of observations in the training and test sets by class,

as well as the length of each for verification.

'''

for k,v in data\_dict.items():

print(k,len(v),len(v[0]))

print()

for k,v in test\_dict.items():

print(k,len(v),len(v[0]))

'''

Create sample image of data: I used this just to help me visualize the data

'''

create\_large\_image(data\_dict)

'''

Randomize the order of the data

This is a hold-over from the NN data handling, but randomizing the data made it

simpler for the to test individual observations of various characters while

prototyping, so I kept it.

'''

data\_array,data\_result = randomize\_data\_arrays(data\_dict)

test\_array,test\_result = randomize\_data\_arrays(test\_dict)

'''

Reshaping the data for the Naive Bayes classifier

The data needed to be rotated in order to reduce the loops needed to train.

'''

data\_reshaped = reshape\_data(data\_array)

test\_reshaped = reshape\_data(test\_array)

'''Instantiating the naive bayes Python Class object'''

nb\_clf = naive\_bayes(data\_reshaped,data\_result)

''' A loop to run the Naive Bayes classifier on the test data'''

correct = 0

correct\_dict = {}

totals\_dict = {}

tested = len(test\_result)

for i in range(len(test\_result)):

if test\_result[i] not in totals\_dict:

totals\_dict[test\_result[i]]=1

else:

totals\_dict[test\_result[i]]+=1

result = nb\_clf.classify(test\_array[i])

output = result.index(max(result))

if output == test\_result[i]:

correct +=1

if output not in correct\_dict:

correct\_dict[output] = 1

else:

correct\_dict[output] += 1

#%%

x = []

y = []

yerr = []

'''Printing the output of the Naive Bayes classifier'''

print("\nOut of",tested,"tests,",correct,"were correct, a ratio of",str(correct/tested)+".\n")

print("Class Total Correct Ratio 95% Confidence Interval")

for k,v in sorted(totals\_dict.items()):

ratio = correct\_dict[k]/v

interval = 1.96\*math.sqrt((ratio\*(1-ratio))/v)

print("%5d %5d %5d %f %f-to-%f"%\

(k,v,correct\_dict[k],ratio,ratio-interval,ratio+interval))

x.append(k)

y.append(ratio)

yerr.append(interval)

print()

'''Print a visualization of the resulting errors'''

plt.errorbar(x, y, yerr=yerr,fmt='o')

plt.ylim(0.6,1)

plt.xticks(x)

#plt.legend(loc='lower right')

plt.show()