**Jericho McLeod**

CSI-873

Homework 7

Table of Contents

[Problem 1 – Bayesian Network Results Discussion 1](#_Toc22580012)

[Problem 2 – EM Algorithm Results Discussion 3](#_Toc22580013)

[Appendix 1 – Bayesian Network Code 6](#_Toc22580014)

[Appendix 2 – EM Algorithm Code 14](#_Toc22580015)

# Problem 1 – Bayesian Network Results Discussion

Using a Bayesian belief network in place of a Naïve Bayes classifier provided some marginal improvements in my model. Overall accuracy increased 2%, from 84% to 86%. What is most notable about Bayesian Belief Networks in comparison to Naïve Bayes is not improved overall accuracy, but rather, more consistent accuracy within the model. The range of accuracies across digits is 13.8% in the Bayesian network, while it is 22.3% in the Naïve model.

The reason for this is attributable to the difference between the models. Naïve Bayes takes into account each pixel equally, while the Bayesian Network, using the configuration of pixels 2 through N considering their immediate predecessor, looks at the points where horizontal changes occur. In other words, it looks at the average left-right side outlines of digits, rather than the intermediary pixels.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Naïve Bayes Classification** | | | | | | | | | |  |
|  | # | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| **Actual Class** | **0** | **455** | 0 | 3 | 5 | 1 | 23 | 7 | 0 | 6 | 0 | 91.0% |
| **1** | 0 | **543** | 5 | 3 | 0 | 3 | 1 | 0 | 12 | 1 | 95.6% |
| **2** | 8 | 2 | **428** | 17 | 10 | 2 | 12 | 7 | 29 | 1 | 82.9% |
| **3** | 3 | 10 | 15 | **420** | 0 | 7 | 7 | 7 | 20 | 16 | 83.2% |
| **4** | 1 | 4 | 3 | 1 | **394** | 2 | 11 | 0 | 9 | 66 | 80.2% |
| **5** | 8 | 9 | 5 | 63 | 12 | **327** | 4 | 4 | 8 | 6 | 73.3% |
| **6** | 9 | 11 | 10 | 2 | 7 | 12 | **425** | 0 | 3 | 0 | 88.7% |
| **7** | 1 | 11 | 4 | 3 | 5 | 2 | 0 | **435** | 12 | 41 | 84.6% |
| **8** | 6 | 15 | 8 | 36 | 8 | 8 | 2 | 3 | **382** | 19 | 78.4% |
| **9** | 4 | 4 | 4 | 2 | 45 | 5 | 0 | 12 | 12 | **417** | 82.6% |
|  |  | 91.9% | 89.2% | 88.2% | 76.1% | 81.7% | 83.6% | 90.6% | 92.9% | 77.5% | 73.5% |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Bayesian Belief Network Classification** | | | | | | | | | |  |
|  | # | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |  |
| **Actual Class** | **0** | **457** | 0 | 2 | 2 | 0 | 6 | 17 | 0 | 5 | 1 | 93.3% |
| **1** | 0 | **530** | 19 | 4 | 0 | 6 | 1 | 0 | 8 | 0 | 93.3% |
| **2** | 21 | 7 | **432** | 14 | 2 | 1 | 9 | 7 | 20 | 3 | 83.7% |
| **3** | 3 | 5 | 11 | **443** | 1 | 14 | 4 | 3 | 13 | 8 | 87.7% |
| **4** | 0 | 3 | 5 | 4 | **412** | 5 | 7 | 2 | 10 | 43 | 83.9% |
| **5** | 6 | 7 | 7 | 41 | 1 | **372** | 2 | 1 | 6 | 3 | 83.4% |
| **6** | 11 | 5 | 7 | 2 | 8 | 12 | **429** | 0 | 5 | 0 | 89.6% |
| **7** | 0 | 10 | 8 | 8 | 4 | 0 | 1 | **429** | 8 | 46 | 83.5% |
| **8** | 9 | 4 | 14 | 30 | 9 | 10 | 2 | 6 | **387** | 16 | 79.5% |
| **9** | 8 | 4 | 9 | 2 | 26 | 1 | 0 | 16 | 11 | **428** | 84.8% |
|  |  | 88.7% | 92.2% | 84.0% | 80.5% | 89.0% | 87.1% | 90.9% | 92.5% | 81.8% | 78.1% |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **95% Confidence Intervals** | | | |
| **Naïve Bayes** | | **Bayesian Network** | |
| **From** | **To** | **From** | **To** |
| 88.5% | 93.5% | 91.0% | 95.5% |
| 93.9% | 97.3% | 91.3% | 95.4% |
| 79.7% | 86.2% | 80.5% | 86.9% |
| 79.9% | 86.4% | 84.9% | 90.6% |
| 76.7% | 83.8% | 80.7% | 87.2% |
| 69.2% | 77.4% | 80.0% | 86.9% |
| 85.9% | 91.6% | 86.8% | 92.3% |
| 81.5% | 87.7% | 80.3% | 86.7% |
| 74.8% | 82.1% | 75.9% | 83.1% |
| 79.3% | 85.9% | 81.6% | 87.9% |

# Problem 2 – EM Algorithm Results Discussion

The EM algorithm finds the maximum a posteriori estimates for the means of a dataset. The first step I took in understanding this algorithm was the plot the data in a scatter plot using an artificial X axis based on the data index and in a histogram. The histogram informed me of the existing distribution, and the scatterplot merely ensured that the data was randomly ordered; had there been an existing pattern it could have informed later choices.

![A close up of a logo

Description automatically generated]()![A screenshot of a cell phone

Description automatically generated]()

Given that the data appeared to be distributed around 0, I chose initial starting points for μ1 andμ2 between 0 and 1, and then made one of them negative, so that I had a number between -1 and 0, and one between 0 and 1. I then ran the algorithm with a stopping condition of both means changing by less than 0.00001. It converged rather quickly on ﻿0.636707 and ﻿-0.577066. I ran the algorithm several times and verified my results, and in a small monte carlo to see the average solve iterations (14).

Then I verified the results looked appropriate, I distributed them according to their probabilities into my original visualizations. This histogram is very similar to the example in the textbook:

![A screenshot of a cell phone

Description automatically generated]()

While the scatterplot is not particularly useful, other than as an example of how this dataset may look over some arbitrary dimension.

![A screenshot of a cell phone

Description automatically generated]()

To make this visualization more meaningful, I created another function to visualize the probabilities of observed data most likely mean:

﻿![A screenshot of a cell phone

Description automatically generated]()

This also highlights the random aspect of the prior scatterplot, in that by pure probability there is not intermingling of classifications.

# Appendix 1 – Bayesian Network Code

Changes from Naïve Bayes are highlighted

﻿#!/usr/bin/env python3

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

"""

Created on Sun Oct 20 14:05:17 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/nth

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

return(data\_dict)

def reshape\_data(data\_dict):

results = []

sideways = []

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

for i in range(len(v)):

sideways.append(v[i])

results.append(k)

print(len(sideways),len(sideways[0]))

data\_reshaped = []

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

data\_reshaped.append([])

for i in range(len(sideways)):

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

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

return(data\_reshaped,results)

denom = 1

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

denom = 2

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

data\_matrix,results\_array = reshape\_data(data\_dict)

test\_matrix,test\_results\_array = reshape\_data(test\_dict)

#%%

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,prior,array,target):

if prior == 0:

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

else:

pos\_count\_dict\_0,pos\_count\_dict\_1,neg\_count\_dict\_0,neg\_count\_dict\_1 = {},{},{},{}

count\_0,count\_1 = 0,0

if prior == 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")

else:

for i in self.outcomes:

pos\_count\_dict\_0[i],pos\_count\_dict\_1[i],neg\_count\_dict\_0[i],neg\_count\_dict\_1[i]=0,0,0,0

for i in range(len(array)):

if array[i] == 0:

if prior[i]==0:

neg\_count\_dict\_0[target[i]]+=1

else:

neg\_count\_dict\_1[target[i]]+=1

count\_0+=1

else:

if prior[i]==0:

pos\_count\_dict\_0[target[i]]+=1

else:

pos\_count\_dict\_1[target[i]]+=1

count\_1+=1

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

'''This is an altered version of the function that converts

probabilities of the evidence based on the prior observations'''

count\_11,count\_01,total\_count = 0,0,0

for i in range(len(array)):

total\_count +=1

if prior[i] == 1:

if array[i] == 1:

count\_11 +=1

elif prior[i] == 0:

if array[i] == 1:

count\_01 +=1

self.pixel\_proba = [count\_01/total\_count,count\_11/total\_count]

'''

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\_0.items():

#try:

try:

proba\_0 = pos\_count\_dict\_0[k] / (pos\_count\_dict\_0[k]+ neg\_count\_dict\_0[k])

except:

proba\_0 = 0

try:

proba\_1 = pos\_count\_dict\_1[k] / (pos\_count\_dict\_1[k]+ neg\_count\_dict\_1[k])

except:

proba\_1 = 0

#print(proba\_0,proba\_1)

self.outcome\_proba[k] = [proba\_0,proba\_1]

try:

proba\_0 = neg\_count\_dict\_0[k] / (pos\_count\_dict\_0[k]+ neg\_count\_dict\_0[k])

except:

proba\_0 = 0

try:

proba\_1 = neg\_count\_dict\_1[k] / (pos\_count\_dict\_1[k]+ neg\_count\_dict\_1[k])

except:

proba\_1 = 0

#print(proba\_0,proba\_1)

self.outcome\_proba\_neg[k] = [proba\_0,proba\_1]

#except:

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

class bayesian\_network:

'''

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()

if i == 0:

self.pixels[i].update\_probas(0,data[i],target)

else:

self.pixels[i].update\_probas(data[i-1],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)):

if i == 0:

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)):

#print(len(posteriors))

#print(len(p\_p))

#print(j)

#print(p\_p[j])

try:

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

except:

print('posteriors:',posteriors,'\n','p\_p:',p\_p,'\n','j:',j)

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

else:

'''This is a new step to determine the probability of the

evidence based on the prior evidence's observed value.

Uncomment the following print lines to see that the probability

of the evidence is being updated'''

#print(array[i],array[i-1],self.pixels[i].pixel\_proba)

p\_e = self.pixels[i].pixel\_proba[int(array[i-1])]

#print(p\_e)

if array[i] == 1:

if array[i-1] == 0:

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

else:

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

else:

if array[i-1] == 0:

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

else:

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

p\_e = 1-p\_e

for j in range(len(posteriors)):

try:

if array[i-1] == 0:

posteriors[j] = posteriors[j] \* p\_p[j][0]

else:

posteriors[j] = posteriors[j] \* p\_p[j][1]

except:

print('posteriors:',posteriors,'\n','p\_p:',p\_p,'\n','j:',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)

bn\_clf = bayesian\_network(data\_matrix,results\_array)

#%%

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)

test\_data = reshape\_data(test\_matrix)

#%%

confusion\_matrix = [[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],\

[0,0,0,0,0,0,0,0,0,0],]

correct = 0

incorrect = 0

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

actual = test\_results\_array[i]

class\_proba = bn\_clf.classify(test\_data[i])

classified\_as = class\_proba.index(max(class\_proba))

confusion\_matrix[int(actual)][(classified\_as)]+=1

if actual == classified\_as:

correct+=1

else:

incorrect+=1

#%%

for i in confusion\_matrix:

print(i)

print()

print(correct/(correct+incorrect))

# Appendix 2 – EM Algorithm Code

﻿#!/usr/bin/env python3

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

"""

Created on Sun Oct 20 15:43:34 2019

@author: jmcleod

"""

import csv,random, math

import matplotlib.pyplot as plt

'''

Implement EM algorithm to estimate 2 means, mu\_1 and mu\_2, for normal distributions

with sigma = 1. Assume that an equal number of points belongs to each distribution.

Include code and results with report.

'''

#%%

data = []

with open('McLeod.txt') as txtfile:

txt\_r = csv.reader(txtfile,delimiter=' ')

for r in txt\_r:

data.append(r)

#%%

data\_float = []

counter = 0

for i in data:

for j in i:

try:

data\_float.append(float(j))

except:

print('error converting ->',j,'<- input.')

mixdata = data\_float

#%%

for i in range(len(mixdata)):

plt.scatter(float(i),mixdata[i])

plt.show()

#%%

mu\_1,mu\_2 = random.random(),-random.random()

h = [mu\_1,mu\_2]

sigma = 1

def calc(sigma,mu,val):

out = math.e\*\*(-(1/(2\*(sigma\*\*2)))\*((mu-val)\*\*2))

return(out)

def em\_step1(h,sigma,data):

data\_expanded = []

for i in data:

x\_1 = calc(sigma,h[0],i)

x\_2 = calc(sigma,h[1],i)

e\_1 = x\_1 / (x\_1+x\_2)

e\_2 = x\_2 / (x\_1+x\_2)

data\_expanded.append([i,e\_1,e\_2])

return(data\_expanded)

def em\_step2(data):

h1,h2 = [0,0],[0,0]

for i in data:

h1[0] += i[0]\*i[1]

h1[1] += i[1]

h2[0] += i[0]\*i[2]

h2[1] += i[2]

h = [h1[0]/h1[1],h2[0]/h2[1]]

return(h)

#%%

def print\_colors\_initial(data):

rgb = (.5,.5,.5)

for i in range(len(data)):

plt.scatter(float(i),data[i],color=[rgb],marker='.')

plt.show()

def print\_colors\_expanded(data\_expanded):

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

r = random.random()

if r > data\_expanded[i][1]:

rgb = (0,0,1)

else:

rgb = (1,0,0)

plt.scatter(float(i),data\_expanded[i][0],color=[rgb],marker='.')

plt.show()

print\_colors\_initial(mixdata)

print(h)

stop = 0

h\_old = h

counter = 0

while stop == 0:

data\_expanded = em\_step1(h,sigma,mixdata)

#if counter == 0:

#print\_colors\_expanded(data\_expanded)

h\_new = em\_step2(data\_expanded)

if h\_new[0]-h\_old[0] < 0.00001:

if h\_new[1]-h\_old[1] < 0.00001:

stop+=1

h\_old = h

h = h\_new

counter+=1

#%%

print(h)

print\_colors\_expanded(data\_expanded)

#%%

solve\_iters = 0

n = 100

for l in range(n):

mu\_1,mu\_2 = random.random(),-random.random()

h = [mu\_1,mu\_2]

sigma = 1

stop = 0

h\_old = h

counter = 0

while stop == 0:

data\_expanded = em\_step1(h,sigma,mixdata)

#if counter == 0:

#print\_colors\_expanded(data\_expanded)

h\_new = em\_step2(data\_expanded)

if h\_new[0]-h\_old[0] < 0.00001:

if h\_new[1]-h\_old[1] < 0.00001:

stop+=1

h\_old = h

h = h\_new

counter+=1

solve\_iters+=counter

print(solve\_iters/n)

#%%

x = []

y = []

h1\_lab = str(h[0])[0:4]

h2\_lab = str(h[1])[0:4]

for i in data\_expanded:

r = random.random()

if r<i[1]:

x.append(i[0])

else:

y.append(i[0])

plt.hist([mixdata],bins=50,label=['data'])

plt.show()

plt.figure(figsize=(12,6))

plt.hist([mixdata,x,y],bins=50,label=['All',h1\_lab,h2\_lab])

plt.legend(loc='upper left')

plt.show()

#%%

def print\_colors\_proba(data\_expanded):

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

min\_value = min(data\_expanded[i][1],data\_expanded[i][2])

rgb = (data\_expanded[i][1],min\_value,data\_expanded[i][2])

plt.scatter(float(i),data\_expanded[i][0],color=[rgb],marker='.')

plt.show()

print\_colors\_proba(data\_expanded)