Huajian Lin AMO student

**Part 1 Accuracies:**

| Setup | Cross-validation Accuracy |
|---|---|
| Unprocessed Data | 76.27% |
| 0-value elements ignored | 74.10% |

## Part 1 Code Snippets

### 1. Calculation of distribution parameters

```
for i in range(0,featureCount-1):
    diaParas[i] = [np.mean(diabeteSet[i]), np.std(diabeteSet[i])]
    nonDiaParas[i] = [np.mean(nonDiabeteSet[i]), np.std(nonDiabeteSet[i])]
pDiabetes = len(diabeteSet[0])/len(dataSet)
diaParas[featureCount-1].append(pDiabetes)
nonDiaParas[featureCount-1].append(1-pDiabetes)
```

*Note* for detail see method:

```
def getNormPara(dataSet, diaParas, nonDiaParas, featureCount, omit = False)
```

### 2. Calculation of naïve Bayes predictions

```
for i in range(0, featureCount):
	if( not (omit and (i in omitCols) and dataPoint[i]==0) ):
		pDia += np.log(norm.pdf(dataPoint[i], diaParas[i][0], diaParas[i][1]))
		pNonDia += np.log(norm.pdf(dataPoint[i], nonDiaParas[i][0],
nonDiaParas[i][1]))
	if( (pDia + np.log(diaParas[featureCount])) > (pNonDia +
np.log(nonDiaParas[featureCount])) ):
		return diabetePositive
	else:
		return diabeteNegative
```

*Note* for detail see method:

```
def predict(dataPoint, diaParas, nonDiaParas, omit = False)
```

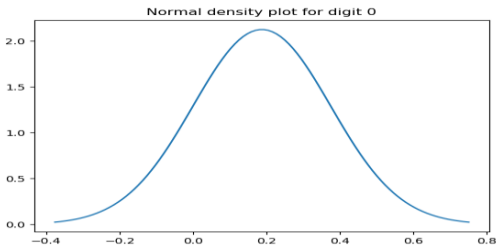
### 3. Test-Train Split Code

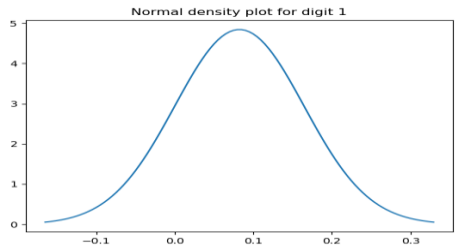
```
def splitData(pTest, test, train, dataSet):
	for d in dataSet:
		#random.random() return a random number between (0,1]
		randPTest = random.random()
		if(randPTest <= pTest):
			test.append(d)
		else:
			train.append(d)
```

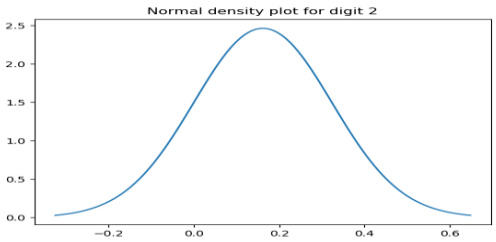
Part 2 MNIST Accuracies:

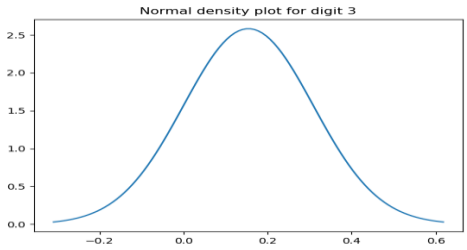
| x | Method | Training Set Accuracy | Test Set Accuracy |
|---|---|---|---|
| 1 | Gaussian + untouched | 62.47% | 62.16% |
| 2 | Gaussian + stretched | 82.09% | 82.97% |
| 3 | Bernoulli + untouched | 83.81% | 84.40% |
| 4 | Bernoulli + stretched | 81.52% | 82.76% |
| 5 | 10 trees + 4 depth + untouched | 69.38% | 69.83% |
| 6 | 10 trees + 4 depth + stretched | 71.78% | 73.1% |
| 7 | 10 trees + 16 depth + untouched | 99.18% | 94.34% |
| 8 | 10 trees + 16 depth + stretched | 99.56% | 94.69% |
| 9 | 30 trees + 4 depth + untouched | 75.17% | 76.30% |
| 10 | 30 trees + 4 depth + stretched | 75.03% | 75.76% |
| 11 | 30 trees + 16 depth + untouched | 99.55% | 95.82% |
| 12 | 30 trees + 16 depth + stretched | 99.76% | 96.39% |

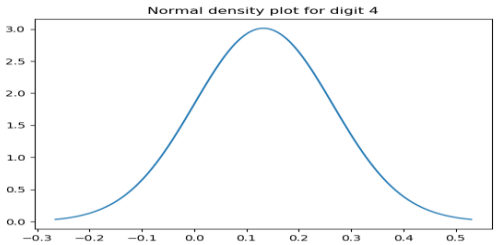
## Part 2A Digit Images:

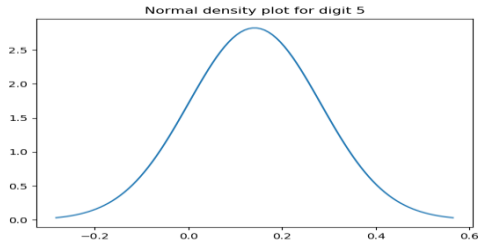


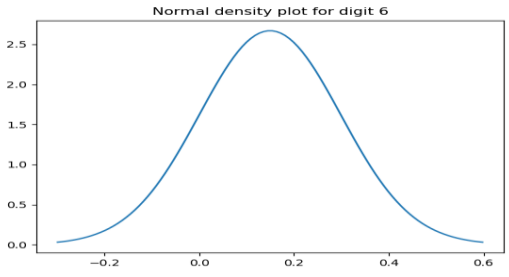



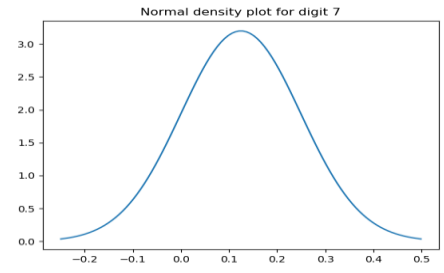



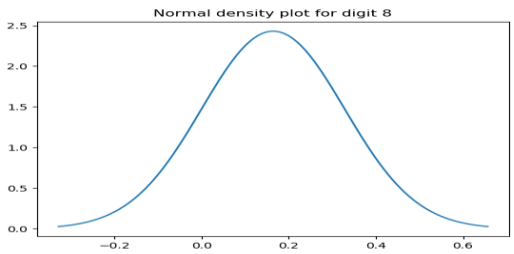



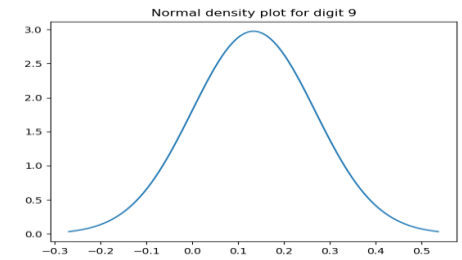



**Part 2 Code:**

**Calculation of the normal distribution parameters:**

```
paras[i] = (np.mean(classMat[i], axis = 0), np.std(classMat[i], axis = 0))
```

*Note* for detail see method:

```
def getParas(imgs, labels, classCount, paras, bern = False):
```

**Calculation of the Bernoulli distribution parameters:**

```
paras[i] = np.mean(classMat[i], axis = 0)
```

*Note* for detail see method:

```
def getParas(imgs, labels, classCount, paras, bern = False):
```

**Calculation of the Naïve Bayes predictions:**

```
for classIndex in range(0, len(classParas)):
	if(bern):
		resultSet[classIndex] = np.nansum(np.log(bernoulli.pmf(img,
paras[classIndex])))
	else:
		resultSet[classIndex] = np.nansum(np.log(norm.pdf(img,
paras[classIndex][0], paras[classIndex][1])))
for i in range(0, len(classParas)):
	resultSet[i] += np.log(classParas[i])
return np.argmax(resultSet)
```

**Training of a decision tree:**

```
clf = RandomForestClassifier(n_estimators = treeNumber, max_depth = maxDepth)
clf.fit(trainImgsOriOneD, trainLabels)
```

*Note* for detail see:

Complete code for Part 2: Forest Classifier

**Calculation of a decision tree predictions:**

```
clfTrainlabels = clf.predict(trainImgsOriOneD)
clfTestlabels = clf.predict(testImgsOriOneD)
```

*Note* for detail see:

Complete code for Part 2: Forest Classifier

Complete code for part 1:

```
import csv
import numpy as np
import random
from scipy.stats import norm

#define some constant
diabetePositive = 1
diabeteNegative = 0
omitCols = [2,3,5,7]

#read and store data
def readCSV(file, dataSet):
	with open(file) as csv_file:
		csv_reader = csv.reader(csv_file)
		for line in csv_reader:
			dataSet.append([float(attr) for attr in line])


#split data into test and train with probability of pTest picking a datapoint for testset
def splitData(pTest, test, train, dataSet):
	for d in dataSet:
		#random.random() return a random number between (0,1]
		randPTest = random.random()
		if(randPTest <= pTest):
			test.append(d)
		else:
			train.append(d)

#find normal distribution parameter for given data
def getNormPara(dataSet, diaParas, nonDiaParas, featureCount, omit = False):
	diabeteSet, nonDiabeteSet = [],[]
	for i in range(0, featureCount):
		diabeteSet.append([])
		nonDiabeteSet.append([])
	#group same features into label group
	for dataPoint in dataSet:
		for i in range(0,featureCount):
			if( not (omit and (i in omitCols) and dataPoint[i]==0) ):
				if(dataPoint[featureCount-1] == diabetePositive):
					diabeteSet[i].append(dataPoint[i])
				else:
					nonDiabeteSet[i].append(dataPoint[i])
	for i in range(0,featureCount-1):
		diaParas[i] = [np.mean(diabeteSet[i]), np.std(diabeteSet[i])]
		nonDiaParas[i] = [np.mean(nonDiabeteSet[i]), np.std(nonDiabeteSet[i])]
	pDiabetes = len(diabeteSet[0])/len(dataSet)
	diaParas[featureCount-1].append(pDiabetes)
	nonDiaParas[featureCount-1].append(1-pDiabetes)

#predict datapoint using given normal distribution parameters
def predict(dataPoint, diaParas, nonDiaParas, omit = False):
	pDia,pNonDia = 0,0
	featureCount = len(dataPoint)-1
	for i in range(0, featureCount):
		if( not (omit and (i in omitCols) and dataPoint[i]==0) ):
```

```
                        pDia += np.log(norm.pdf(dataPoint[i], diaParas[i][0],
diaParas[i][1]))
                        pNonDia += np.log(norm.pdf(dataPoint[i], nonDiaParas[i][0],
nonDiaParas[i][1]))
        if( (pDia + np.log(diaParas[featureCount])) > (pNonDia +
np.log(nonDiaParas[featureCount])) ):
                return diabetePositive
        else:
                return diabeteNegative

#set up data
dataSet = []
file = 'pima-indians-diabetes.csv'
readCSV(file, dataSet)
featureCount = len(dataSet[0])

#perform train and test
numSplit = 10
pTest = 0.2
accuracy = []
for i in range(0, numSplit):
        test, train, diaParas, nonDiaParas = [],[],[],[]
        for i in range(0, featureCount):
                diaParas.append([])
                nonDiaParas.append([])
        #split data
        splitData(pTest, test, train, dataSet)
        #get distribution parameter on train set
        getNormPara(train, diaParas, nonDiaParas, featureCount)
        #make prediction of test set
        correctCount = 0
        for t in test:
                result = predict(t, diaParas, nonDiaParas)
                if(result == t[len(t)-1]):
                        correctCount += 1
        accuracy.append(correctCount/len(test))

for a in accuracy:
        print(a)
print(f'Average accuracy is : {np.mean(accuracy)}')

#perform train and test while omitting zero value for 3rd, 4th, 6th and 8th column
accuracy = []
for i in range(0, numSplit):
        test, train, diaParas, nonDiaParas = [],[],[],[]
        for i in range(0, featureCount):
                diaParas.append([])
                nonDiaParas.append([])
        #split data
        splitData(pTest, test, train, dataSet)
        #get distribution parameter on train set
        getNormPara(train, diaParas, nonDiaParas, featureCount, True)
        #make prediction of test set
        correctCount = 0
        for t in test:
                result = predict(t, diaParas, nonDiaParas, True)
                if(result == t[len(t)-1]):
                        correctCount += 1
```

```
        accuracy.append(correctCount/len(test))

for a in accuracy:
        print(a)
print(f'Average accuracy after omitting 0 value for column 3,4,6,8 is :
{np.mean(accuracy)}')
```

### Complete code for part 2:

### Prepare Data:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
from scipy.stats import norm
from scipy.stats import bernoulli
from sklearn.ensemble import RandomForestClassifier

#thresholding image, for any value less than threshold, set to min, else set to max
def thresholding(imgs, threshold, min, max):
        for img in imgs:
                for row in range(0, len(img)):
                        low = img[row] < threshold
                        high = img[row] >= threshold
                        img[row][low] = min
                        img[row][high] = max

#convert all arrays to matrixs with given dimension
def arrayToMatrix(src, tar, dim, reduceDim = False):
        for arr in src:
                if(reduceDim):
                        tar.append(np.reshape(arr, dim)[0])
                else:
                        tar.append(np.reshape(arr, dim))

#get min,max value of non-zero row index
def getRowMinMax(img):
        min, max = len(img)//2, len(img)//2
        for row in range(0,len(img)):
                if(np.max(img[row]) != 0):
                        if(row < min):
                                min = row
                        if(row > max):
                                max = row
        return min,max

#first shrink image to get rid of all zero row and column, then resized back to resizedDim
def shrinkAndResize(imgs, touchedImgs, resizeDim):
        for img in imgs:
                yMin, yMax = getRowMinMax(img)
                xMin, xMax = getRowMinMax(img.T)
                shrinkImg = img[yMin:yMax+1].T[xMin:xMax+1].T
                resizeImg = cv2.resize(shrinkImg, resizeDim,
interpolation=cv2.INTER_NEAREST)
                touchedImgs.append(resizeImg)
```

```
#find normal or bernoulli distribution parameter
def getParas(imgs, labels, classCount, paras, bern = False):
	classMat = []
	for i in range(0, classCount):
		paras.append([])
		classMat.append([])
	for i in range(0, len(imgs)):
		classMat[labels[i]].append(imgs[i])
	for i in range(0,classCount):
		if(bern):
			paras[i] = np.mean(classMat[i], axis = 0)
		else:
			paras[i] = (np.mean(classMat[i], axis = 0), np.std(classMat[i],
axis = 0))

#get class probability
def getClassParas(labels, classCount, classParas):
	for i in range(0, classCount):
		classParas.append(0)
	for label in labels:
		classParas[label] += 1
	for i in range(0, classCount):
		classParas[i] /= len(labels)

#predict base on bayes parameters and return class label
def predict(img, paras, classParas, bern = False):
	resultSet = []
	for i in range(0, len(classParas)):
		resultSet.append(0)
	for classIndex in range(0, len(classParas)):
		if(bern):
			resultSet[classIndex] = np.nansum(np.log(bernoulli.pmf(img,
paras[classIndex])))
		else:
			resultSet[classIndex] = np.nansum(np.log(norm.pdf(img,
paras[classIndex][0], paras[classIndex][1])))
	for i in range(0, len(classParas)):
		resultSet[i] += np.log(classParas[i])
	return np.argmax(resultSet)


#load data
mndata = MNIST()
mndata.gz = True
trainImgs, trainLabels = mndata.load_training()
testImgs, testLabels = mndata.load_testing()

#set up constant
original, stretched = 0, 1
dataToUse = [original, stretched]
threshold = 100
oriDim = (28,28)
stretchDim = (20,20)
classCount = 10
normalMin = 0
normalMax = 255
beruMin = 0
```

```
beruMax = 1

#set up storage
trainImgsOri, testImgsOri, trainImgsTouched, testImgsTouched = [],[],[],[]

#convert to matrix
arrayToMatrix(trainImgs, trainImgsOri, oriDim)
arrayToMatrix(testImgs, testImgsOri, oriDim)

#thresholding matrix
thresholding(trainImgsOri, threshold, beruMin, beruMax)
thresholding(testImgsOri, threshold, beruMin, beruMax)

#shrink image and then stretch image
shrinkAndResize(trainImgsOri, trainImgsTouched, stretchDim)
shrinkAndResize(testImgsOri, testImgsTouched ,stretchDim)
```

**NaiveBayes using Bernoulli:**

```
#thresholding matrix
thresholding(trainImgsOri, threshold, beruMin, beruMax)
thresholding(testImgsOri, threshold, beruMin, beruMax)

#shrink image and then stretch image
shrinkAndResize(trainImgsOri, trainImgsTouched, stretchDim)
shrinkAndResize(testImgsOri, testImgsTouched ,stretchDim)

#train for bernoulli and original data
paras, classParas = [],[]
getParas(trainImgsOri, trainLabels, classCount, paras, bern = True)
getClassParas(trainLabels, classCount, classParas)
correct = 0
for i in range(0,len(trainImgsOri)):
       if(predict(trainImgsOri[i],paras,classParas, bern = True)== trainLabels[i]):
              correct += 1
print(f'bernoulli untouched using train: {correct / len(trainLabels)}')

correct = 0
for i in range(0,len(testImgsOri)):
       if(predict(testImgsOri[i],paras,classParas, bern = True) == testLabels[i]):
              correct += 1
print(f'bernoulli untouched using test: {correct / len(testLabels)}')

#train for bernoulli and streched data
paras, classParas = [],[]
getParas(trainImgsTouched, trainLabels, classCount, paras, bern = True)
getClassParas(trainLabels, classCount, classParas)
correct = 0
for i in range(0,len(trainImgsTouched)):
       if(predict(trainImgsTouched[i],paras,classParas, bern = True) ==
trainLabels[i]):
              correct += 1
print(f'bernoulli touched using test: {correct / len(trainLabels)}')

correct = 0
for i in range(0,len(testImgsTouched)):
       if(predict(testImgsTouched[i],paras,classParas, bern = True) ==
testLabels[i]):
```

```
            correct += 1
print(f'bernoulli touched using test: {correct / len(testLabels)}')
```

### Naïve Bayes using Noraml:

```
#thresholding matrix
thresholding(trainImgsOri, threshold, normalMin, normalMax)
thresholding(testImgsOri, threshold, normalMin, normalMax)

#shrink image and then stretch image
shrinkAndResize(trainImgsOri, trainImgsTouched, stretchDim)
shrinkAndResize(testImgsOri, testImgsTouched ,stretchDim)

#train for normal and original data
paras, classParas = [],[]
getParas(trainImgsOri, trainLabels, classCount, paras)
getClassParas(trainLabels, classCount, classParas)
correct = 0
for i in range(0,len(trainImgsOri)):
      if(predict(trainImgsOri[i],paras,classParas) == trainLabels[i]):
            correct += 1
print(f'Normal untouched using train: {correct / len(trainLabels)}')

correct = 0
for i in range(0,len(testImgsOri)):
      if(predict(testImgsOri[i],paras,classParas) == testLabels[i]):
            correct += 1
print(f'Normal untouched using test: {correct / len(testLabels)}')

#train for normal and streched data
paras, classParas = [],[]
getParas(trainImgsTouched, trainLabels, classCount, paras)
getClassParas(trainLabels, classCount, classParas)
correct = 0
for i in range(0,len(trainImgsTouched)):
      if(predict(trainImgsTouched[i],paras,classParas) == trainLabels[i]):
            correct += 1
print(f'Normal touched using test: {correct / len(trainLabels)}')

correct = 0
for i in range(0,len(testImgsTouched)):
      if(predict(testImgsTouched[i],paras,classParas) == testLabels[i]):
            correct += 1
print(f'Normal touched using test: {correct / len(testLabels)}')
```

### Digit Plot using pyplot:

```
import matplotlib.pyplot as plt

#plot normal distribution graph for each digit using mean pixel values for each digit
using un-threshold training value
def plotNormalForDigits(imgs, labels, classCount):
      classMat, normalPara = [],[]
      for i in range(0, classCount):
            classMat.append([])
            normalPara.append([])
      for i in range(0, len(imgs)):
            classMat[labels[i]].append(imgs[i])
      for i in range(0,classCount):
```

```
            normalPara[i] = (np.mean(classMat[i]), np.std(classMat[i]))
            x =
np.linspace(normalPara[i][0]-3*normalPara[i][0],normalPara[i][0]+3*normalPara[i][0
],100)
            plt.figure(i)
            plt.title(f'Normal density plot for digit {i}')
            plt.plot(x, norm.pdf(x, normalPara[i][0], normalPara[i][0]))
```

**Forest Classifier:**

```
from sklearn.ensemble import RandomForestClassifier

#train using forest
treeNumbers, maxDepths = [10, 30], [4, 16]
trainImgsOriOneD, testImgsOriOneD, trainImgsTouOneD, testImgsTouOneD = [],[],[],[]
oneDDimL, oneDDimS = (1, 28*28), (1, 20*20)
arrayToMatrix(trainImgsOri, trainImgsOriOneD, oneDDimL, reduceDim = True)
arrayToMatrix(testImgsOri, testImgsOriOneD, oneDDimL, reduceDim = True)
arrayToMatrix(trainImgsTouched, trainImgsTouOneD, oneDDimS, reduceDim = True)
arrayToMatrix(testImgsTouched, testImgsTouOneD, oneDDimS, reduceDim = True)


for treeNumber in treeNumbers:
      for maxDepth in maxDepths:
            for data in dataToUse:

                  clf = RandomForestClassifier(n_estimators = treeNumber, max_depth
= maxDepth)
                  clfTrainlabels, clfTestlabels = [],[]
                  trainMessage, testMessage = '',''
                  trainCorrect, testCorrect = 0, 0
                  if(data == original):
                        clf.fit(trainImgsOriOneD, trainLabels)
                        clfTrainlabels = clf.predict(trainImgsOriOneD)
                        clfTestlabels = clf.predict(testImgsOriOneD)
                        trainMessage = f'Accuracy for Forest classifier with
{treeNumber} trees, {maxDepth} maximum Depth, untouched, train data: '
                        testMessage = f'Accuracy for Forest classifier with
{treeNumber} trees, {maxDepth} maximum Depth, untouched, test data: '
                  else:
                        clf.fit(trainImgsTouOneD, trainLabels)
                        clfTrainlabels = clf.predict(trainImgsTouOneD)
                        clfTestlabels = clf.predict(testImgsTouOneD)
                        trainMessage = f'Accuracy for Forest classifier with
{treeNumber} trees, {maxDepth} maximum Depth, touched, train data: '
                        testMessage = f'Accuracy for Forest classifier with
{treeNumber} trees, {maxDepth} maximum Depth, touched, test data: '

                  for i in range(0,len(clfTrainlabels)):
                        if(clfTrainlabels[i] == trainLabels[i]):
                              trainCorrect += 1
                  print(trainMessage + f'{trainCorrect / len(trainLabels)}')

                  for i in range(0,len(clfTestlabels)):
                        if(clfTestlabels[i] == testLabels[i]):
                              testCorrect += 1
                  print(testMessage + f'{testCorrect / len(testLabels)}')
```