Machine Learning Naive Bayes Tutorial

Tamlin Love - 1438243

1.

a)

An list of short, 18 line reviews were passed in the following program, which used a random 12 of the reviews as training data and tested sentiment analysis on the other 6. The training reviews were broken up into words and these words were stored in an array, *words*. Two other arrays, *positiveCount* and *negativeCount*, were used to the number of occurrences of each word in the positive and negative reviews respectively.

Using Naive-Bayes (and Laplace smoothing where the occurrence of a word for a given sentiment was 0 or the word had not yet been encountered), the probability of a given sentiment (positive or negative) for a testing review's words was calculated.

From this, a confusion matrix was created. The algorithm was run 1000 times and an average confusion matrix was created to better understand the performance of the algorithm.

The code is shown below.

```
import NBControl as nb
import numpy as np
from random import randint
from math import exp

reviewText = nb.readFile('simple-food-reviews.txt')
trainingNo = 12
bigConfusion = np.zeros([2,2])
bigIteration = 1000

for iteration in range(bigIteration):
    reviews = reviewText.split('\n')    #Split text into reviews

    #Generate a random list of 12 reviews for training
    trainingReviews = []
    for i in range(trainingNo):
        j = randint(0,len(reviews)-1)
        a = reviews.pop(j)
```

```
    trainingReviews.append(a)

#Do training
words = []
positiveCount = []
negativeCount = []
positiveNo = 0
negativeNo = 0
totalNo = trainingNo

for i in range(trainingNo):
    thisReviewWords = trainingReviews[i].split()   #Split review into
         its words
    thisSentiment = thisReviewWords[0]             #Take sentiment
    if thisSentiment == "-1":
        negativeNo = negativeNo + 1
    else:
        positiveNo = positiveNo + 1
    for j in range(1,len(thisReviewWords)):
        if thisReviewWords[j] in words:
            index = words.index(thisReviewWords[j])
            if thisSentiment == "-1":
                negativeCount[index] = negativeCount[index] + 1
            else:
                positiveCount[index] = positiveCount[index] + 1
        else:
            words.append(thisReviewWords[j])
            if thisSentiment == "-1":
                negativeCount.append(1)
                positiveCount.append(0)
            else:
                positiveCount.append(1)
                negativeCount.append(0)


#Do testing

PNeg = 1.0*negativeNo/totalNo
PPos = 1.0*positiveNo/totalNo
totalCount = np.array(negativeCount)+np.array(positiveCount)
k = 1
nk = 2
confusion = np.zeros([2, 2])
for i in range(len(reviews)):
    thisReviewWords = reviews[i].split() #Split review into words
    mySentiment = thisReviewWords.pop(0)
    runningProbPos = 1
    runningProbNeg = 1
    for j in range(len(thisReviewWords)):
        if thisReviewWords[j] not in words:
```

```
                runningProbNeg *= 1.0 * k / (negativeNo + nk)
                runningProbPos *= 1.0 * k / (positiveNo + nk)
        for j in range(len(words)):
            if words[j] in thisReviewWords:
                if negativeCount[j]==0:
                    runningProbNeg *= 1.0 * k / (negativeNo + nk)
                else:
                    runningProbNeg *= 1.0 * (negativeCount[j]) /
                         (negativeNo)
                if positiveCount[j]==0:
                    runningProbPos *= 1.0 * k / (positiveNo + nk)
                else:
                    runningProbPos *= 1.0 * (positiveCount[j]) /
                         (positiveNo)
            else:
                runningProbNeg *= 1.0 * (1 - (negativeCount[j] + k) /
                     (negativeNo + nk))
                runningProbPos *= 1.0 * (1 - (positiveCount[j] + k) /
                     (positiveNo + nk))
        print(runningProbNeg)
        negProb = 1.0 * (runningProbNeg * PNeg) / ((runningProbNeg *
             PNeg) + (runningProbPos * PPos))
        posProb = 1.0 * (runningProbPos * PPos) / ((runningProbNeg *
             PNeg) + (runningProbPos * PPos))
        if negProb >= posProb:
            prediction = '-1'
        else:
            prediction = '1'
        if prediction == mySentiment:
            if prediction == '1':
                confusion[0,0] += 1
            else:
                confusion[1,1] += 1
        else:
            if prediction == '1':
                confusion[0,1] += 1
            else:
                confusion[1,0] += 1
    bigConfusion += confusion

bigConfusion = np.divide(bigConfusion,bigIteration)
print(bigConfusion)
```

After the code was run, the average confusion matrix produced looked like this:

	Actually positive	Actually negative
Predicted positive	1.814	1.631
Predicted negative	1.161	1.394

This represents fairly poor accuracy of 53.47%, which can be attributed to the tiny training data size and the over-emphasis on neutral words (such as *the*, *I* and *we*, for example).

b)
The code above was modified to use all 18 reviews for training, while additional reviews were inputted by the user. It was also modified to immediately output the prediction and probability of the sentiment given the review. A -1 would terminate the program. An example output is shown below:

```
Enter a review: the meal was lovely
I am  71 % sure that this review is  positive
Enter a review: the service was terrible
I am  88 % sure that this review is  negative
Enter a review: awful food and a bad experience
I am  67 % sure that this review is  negative
Enter a review: the experience was great
I am  77 % sure that this review is  positive
Enter a review: -1
Bye
```

To confuse the program, the following inputs were given:

```
Enter a review: the meal was not bad
I am  79 % sure that this review is  negative
Enter a review: the meal was a great disappointment
I am  83 % sure that this review is  positive
Enter a review: -1
Bye
```

In the first case, the word *bad* indicated a negative review, which is false as the qualifier *not* negates the word *bad*. Not only did the word *not* not appear in any of the training reviews, but the program was not equipped to deal with a change of meaning created by joining two words (*not bad* is opposite to *bad*).

In the second case, the word *great* indicated a positive review when in fact it merely described the word *disappointment*, which indicates negativity. Once again, not only was *disappointment* not in the training reviews, but the change of meaning when words are combined was not anticipated by the algorithm.

c)
Modifying the code presented in Question 1 (a) to ignore all entries with 0 probability rather than applying Laplace-smoothing to them, the following average confusion matrix was generated:

	Actually positive	Actually negative
Predicted positive	0.763	1.597
Predicted negative	2.121	1.294

This matrix indicates far worse performance (34.28%) than the same algorithm with Laplace smoothing, demonstrating the positive effects of the technique.

d)

After modifying the program to ignore words of 2 characters or less, the following average confusion matrix was produced:

	Actually positive	Actually negative
Predicted positive	1.788	1.386
Predicted negative	1.206	1.62

This matrix indicates moderately better performance (56.80%) than the original one, suggesting that ignoring small words that are unlikely to contribute to the sentiment is a good idea.

2.

The following code was used to perform sentiment analysis on a set of real movie reviews, with 90% of the reviews used as training data and the remaining 10% used as testing data. Laplace smoothing was employed, and words which appeared in testing reviews but not in training reviews were ignored.

The use of a larger dataset was challenging as it forced the code to be optimized for performance. Dictionaries were used instead of simple lists as the time complexity for checking whether a value appears in a dictionary is *O(1)*, as opposed to *O(n)* for a list. Additionally, underflow became an issue when multiplying probabilities. To overcome this, the natural logarithm of each word's probability was taken and then added together. A high precision library, Decimal, was used to convert the logarithms back to the standard form for use in the Naive-Bayes formula.

```
import NBControl as nb
import numpy as np
from random import randint
from math import ceil
from math import log
from decimal import *

reviewText = nb.readFile('movie-pang02.csv')   #   Read reviews from file

#   Settings

trainingPercent = 0.9    #Percentage of reviews to be used for training
getcontext().prec = 100  #Precision of decimals

k = 1             #Laplace Smoothing Numerator
nk = 2            #Laplace Smoothing Denominator

bigConfusion = np.zeros([2, 2])
bigIter = 60

for iteration in range(bigIter):
    #   Generate list of reviews

    allReviews = reviewText.split('\n') #Split text into reviews
    allReviews.pop(0)           #Remove table head
    trainingNo = int(ceil(trainingPercent*len(allReviews))) #Exact
        number of reviews for training
    trainingReviews = []
    for i in range(trainingNo):
        j = randint(0, len(allReviews) - 1)
        a = allReviews.pop(j)
        trainingReviews.append(a)

    #   Training
```

```
print 'Training begins on iteration ', iteration+1

posNo = 0     #Number of positive reviews in training set
negNo = 0     #Number of negative reviews in training set
posWords = {} #Dictionary of positive words
negWords = {} #Dictionary of negative words
for i in range(trainingNo):
    #print 'Training Review #',i+1
    firstSplit = trainingReviews[i].split(',')     #Split the review
        into Sentiment and Text
    thisReviewWordsSet = set(firstSplit[1].split()) # Split review
        into a set of unique words
    thisReviewWords = list(thisReviewWordsSet)     # Create a list
        from the set
    mySentiment = firstSplit[0]                    #Take sentiment
    if mySentiment == 'Pos':
        posNo += 1
    else:
        negNo += 1
    for j in range(len(thisReviewWords)):
        negMod = 0
        posMod = 0
        if mySentiment == 'Pos':
            posMod = 1
        else:
            negMod = 1
        if thisReviewWords[j] in posWords:
            posWords[thisReviewWords[j]] += posMod
        else:
            posWords[thisReviewWords[j]] = posMod
        if thisReviewWords[j] in negWords:
            negWords[thisReviewWords[j]] += negMod
        else:
            negWords[thisReviewWords[j]] = negMod

#   Testing
print 'Testing begins on iteration ', iteration+1

confusion = np.zeros([2, 2])
posProb = 1.0*posNo/trainingNo
negProb = 1.0*negNo/trainingNo

for i in range(len(allReviews)):
    #print 'Testing Review #',i+1
    firstSplit = trainingReviews[i].split(',')     # Split the review
        into Sentiment and Text
    thisReviewWordsSet = set(firstSplit[1].split()) # Split review
        into a set of unique words
    thisReviewWords = list(thisReviewWordsSet)     #Create a list
```

```
            from the set
        mySentiment = firstSplit[0]                    # Take sentiment
        runningNeg = 0
        runningPos = 0
        wordCount = 0
        for j in posWords:
            if j in thisReviewWords:
                wordCount += 1
                runningPos += log(posWords[j]+k)-log(posNo+nk)
                runningNeg += log(negWords[j]+k)-log(negNo+nk)
            else:
                runningPos += log(1 - 1.0*(posWords[j]+k)/(posNo+nk))
                runningNeg += log(1 - 1.0*(negWords[j]+k)/(negNo+nk))

        #For this, we ignore words that are in review i but not in the
            dictionary, as the effect is minimal

        PNegPart = Decimal.exp(Decimal(runningNeg))*Decimal(negProb)
        PPosPart = Decimal.exp(Decimal(runningPos))*Decimal(posProb)
        PNeg = PNegPart/(PNegPart+PPosPart)
        PPos = PPosPart/(PNegPart+PPosPart)
        if PNeg >= PPos:
            prediction = 'Neg'
        else:
            prediction = 'Pos'
        if prediction == mySentiment:
            if prediction == 'Pos':
                confusion[0, 0] += 1
            else:
                confusion[1, 1] += 1
        else:
            if prediction == 'Pos':
                confusion[0, 1] += 1
            else:
                confusion[1, 0] += 1

    print 'Finished with iteration ', iteration+1
    bigConfusion += confusion
    intermediateConfusion = np.divide(bigConfusion, iteration+1)
    print 'Current confusion matrix: '
    print intermediateConfusion

bigConfusion = np.divide(bigConfusion, bigIter)
print 'Final confusion matrix: '
print bigConfusion
```

The algorithm was repeated 60 times to produce an average confusion matrix, shown below.

	Actually positive	Actually negative
Predicted positive	91.23333333	0.4
Predicted negative	9.55	98.81666667

This corresponds to 95.02% accuracy, which is a very good result. It could potentially be improved by removing words of less than a certain length (for example, removing words of length less than three, as in question 1 (d) above).

3.
The normal probability density function is given as

$$f(x_i|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x_i-\mu)^2}{2\sigma^2}} \tag{1}$$

Now, given the assumption that each sampled observation is independent, we can express the likelihood function of the distribution as

$$\begin{aligned} L(\mu,\sigma^2|\bar{x}) &= \textstyle\prod_{i=1}^n f(x_i|\mu,\sigma^2) \\ &= \textstyle\prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x_i-\mu)^2}{2\sigma^2}} \\ &= (2\pi\sigma^2)^{-\frac{n}{2}}e^{-\frac{1}{2\sigma^2}\sum_{i=1}^n(x_i-\mu)^2} \end{aligned} \tag{2}$$

We can simplify the likelihood function by taking its natural logarithm.

$$\begin{aligned} \ln(L(\mu,\sigma^2|\bar{x})) &= \ln((2\pi\sigma^2)^{-\frac{n}{2}}e^{-\frac{1}{2\sigma^2}\sum_{i=1}^n(x_i-\mu)^2}) \\ &= \ln((2\pi\sigma^2)^{-\frac{n}{2}}) + \ln(e^{-\frac{1}{2\sigma^2}\sum_{i=1}^n(x_i-\mu)^2}) \\ &= -\tfrac{n}{2}[\ln(2\pi)+\ln(\sigma^2)] - \tfrac{1}{2\sigma^2}\textstyle\sum_{i=1}^n(x_i-\mu)^2 \end{aligned} \tag{3}$$

Now, since we are looking for the maximum likelihood, we need to maximize the likelihood function, which is equivalent to maximizing Equation (3). In other words, we must set

$$\nabla \ln(L(\mu,\sigma^2|\bar{x})) = 0 \tag{4}$$

Which of course is equivalent to setting

$$\frac{\partial}{\partial\mu}\ln(L(\mu,\sigma^2|\bar{x})) = 0 \tag{5}$$

and

$$\frac{\partial}{\partial\sigma^2}\ln(L(\mu,\sigma^2|\bar{x})) = 0 \tag{6}$$

Solving first for Equation (5),

$$\begin{aligned}\frac{\partial}{\partial\mu}\ln(L(\mu,\sigma^2|\bar{x})) &= \tfrac{\partial}{\partial\mu}(-\tfrac{n}{2}[\ln(2\pi)+\ln(\sigma^2)] - \tfrac{1}{2\sigma^2}\textstyle\sum_{i=1}^{n}(x_i-\mu)^2) \\ &= \tfrac{1}{\sigma^2}\textstyle\sum_{i=1}^{n}(x_i-\mu)\end{aligned} \tag{7}$$

Since $\frac{1}{\sigma^2} \neq 0$, Equation (5) is only satisfied if

$$\sum_{i=1}^{n}(x_i-\mu) = 0 \tag{8}$$

Thus

$$\sum_{i=1}^{n}(x_i) - n\mu = 0 \tag{9}$$

And so, solving for μ, we arrive at

$$\mu = \frac{1}{n}\sum_{i=1}^{n}(x_i) \tag{10}$$

which is the sample mean.

Similarly, we now solve Equation (6) for σ^2.

$$\begin{aligned}\frac{\partial}{\partial\sigma^2}\ln(L(\mu,\sigma^2|\bar{x})) &= \tfrac{\partial}{\partial\sigma^2}(-\tfrac{n}{2}[\ln(2\pi)+\ln(\sigma^2)] - \tfrac{1}{2\sigma^2}\textstyle\sum_{i=1}^{n}(x_i-\mu)^2) \\ &= \tfrac{\partial}{\partial\sigma^2}(-\tfrac{n}{2}\ln(\sigma^2)) - \tfrac{\partial}{\partial\sigma^2}(\tfrac{1}{2\sigma^2}\textstyle\sum_{i=1}^{n}(x_i-\mu)^2) \\ &= -\tfrac{n}{2\sigma^2} - (\textstyle\sum_{i=1}^{n}(x_i-\mu)^2))\tfrac{\partial}{\partial\sigma^2}\tfrac{1}{2\sigma^2} \\ &= -\tfrac{n}{2\sigma^2} + (\textstyle\sum_{i=1}^{n}(x_i-\mu)^2))\tfrac{1}{2(\sigma^2)^2} \\ &= \tfrac{1}{2\sigma^2}(-n + \tfrac{1}{\sigma^2}\textstyle\sum_{i=1}^{n}(x_i-\mu)^2)) \\ &= 0\end{aligned} \tag{11}$$

Again, $\frac{1}{\sigma^2} \neq 0$. Thus,

$$\frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 - n = 0 \tag{12}$$

And therefore

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2 \tag{13}$$

which is the sample variance.

4.

The following code was used to tell which digit was displayed on an 8 pixel by 8 pixel, binarized image of a digit, using Naive-Bayes with Laplace Smoothing.

```
import NBControl as nb
import numpy as np
from random import randint
from math import ceil
from math import log
from decimal import *

digitText = nb.readFile('smalldigits.csv')   # Read digits from file

#   Settings

trainingPercent = 0.8    # Percentage of digits to be used for training
getcontext().prec = 100  # Precision of decimals

k = 1             # Laplace Smoothing Numerator
nk = 2            # Laplace Smoothing Denominator

bigConfusion = np.zeros([10, 10])
bigIter = 100

for iteration in range(bigIter):
    # Generate list of digits
    allDigits = digitText.split('\n') # Split text into each digit
    trainingNo = int(ceil(trainingPercent * len(allDigits))) # Exact
        number of digits for training
    trainingDigits = []
    for i in range(trainingNo):
        j = randint(0, len(allDigits) - 1)
        a = allDigits.pop(j)
        trainingDigits.append(a)

    # Training
    print 'Training begins on iteration #',iteration
    digitNo = np.zeros(10)      # List of number of occurrence of each
        digit
    pixelMaps = []              # Create a list of arrays, each one of
        which will list occurences of pixels for a given digit
    for i in range(10):
        pixelMaps.append(np.zeros(64))

    for i in range(trainingNo):
        # print 'Training digit ',i+1
        thisDigit = trainingDigits[i].split(',')   # Split digit into
            individual pixels (64) + digit
        myDigit = int(thisDigit.pop(64))           # Get actual digit
        thisDigit = np.asarray(np.array(thisDigit),int)
```

```
    digitNo[myDigit] += 1
    pixelMaps[myDigit] += thisDigit             # Add this digit's
        array to the count array

# Testing
print 'Testing begins on iteration #',iteration

confusion = np.zeros([10,10])
digitProbs = []
for i in range(10):
    digitProbs.append(1.0*digitNo[i]/trainingNo)

for i in range(len(allDigits)):
    # print 'Testing digit ',i+1
    thisDigit = trainingDigits[i].split(',')       # Split digit into
        individual pixels (64) + digit
    myDigit = int(thisDigit.pop(64))               # Get actual digit
    runnings = np.zeros(10)
    for j in range(len(thisDigit)):
        if thisDigit[j] == '1':
            for m in range(10):
                runnings[m] +=
                    log(pixelMaps[m][j]+k)-log(digitNo[m]+nk)
        else:
            for m in range(10):
                runnings[m] += log(1 -
                    1.0*(pixelMaps[m][j]+k)/(digitNo[m]+nk))

    probPart = []
    denom = 0
    for j in range(10):
        number =
            Decimal.exp(Decimal(runnings[j]))*Decimal(digitProbs[j])
        probPart.append(number)
        denom += number

    myProb = []
    for j in range(10):
        myProb.append(probPart[j]/denom)

    prediction = myProb.index(max(myProb))
    confusion[prediction,myDigit] += 1

print 'Finished with iteration ',iteration+1
bigConfusion += confusion
intermediateConfusion = np.divide(bigConfusion, iteration + 1)
print 'Current confusion matrix: '
print intermediateConfusion
accuracy = 0
for j in range(10):
```

```
        accuracy += intermediateConfusion[j, j]
    accuracy = 1.0 * accuracy / len(allDigits)
    print 'Current Accuracy = ', 100 * accuracy, '%'


bigConfusion = np.divide(bigConfusion, bigIter)
print 'Final confusion matrix: '
print bigConfusion
accuracy = 0
for j in range(10):
    accuracy += bigConfusion[j, j]
accuracy = 1.0*accuracy/len(allDigits)
print 'Accuracy = ', 100*accuracy, '%'
```

The algorithm was run 100 times to produce the following average confusion matrix.

	0	1	2	3	4	5	6	7	8	9
Predicted 0	35.12	0.00	0.00	0.25	0.00	0.00	0.12	0.00	0.00	0.00
Predicted 1	0.00	26.58	1.88	0.50	0.36	0.24	0.70	0.00	2.66	1.43
Predicted 2	0.00	3.45	31.60	0.54	0.00	0.00	0.00	0.00	0.19	0.15
Predicted 3	0.00	0.00	0.68	30.88	0.00	0.01	0.00	0.00	0.28	1.44
Predicted 4	0.17	0.18	0.00	0.00	34.25	0.38	0.22	0.26	0.00	0.26
Predicted 5	0.21	0.17	0.00	0.42	0.00	32.04	0.01	0.00	0.59	0.67
Predicted 6	0.00	0.36	0.00	0.23	0.00	0.18	34.92	0.00	0.20	0.00
Predicted 7	0.00	0.42	0.22	1.41	1.28	0.20	0.00	34.42	0.41	1.46
Predicted 8	0.07	4.28	0.81	1.17	0.53	0.24	0.27	0.18	28.16	0.51
Predicted 9	0.00	1.59	0.52	1.51	0.00	2.80	0.00	0.00	1.83	29.93

This corresponds to an accuracy of 88.55%, which is very reasonable. A few common confusions include confusing 8 and 1, 2 and 1 and 9 and 3, all of which are likely due to the low resolution of the images. This may be improved by using a larger dataset and by using more detailed images (larger size and non-binarized).