Project 1 Naïve Bayes

COMP4901K and MATH 4824B

Fall 2018

Notes:

- Due: October 18 at 23:59. **No late submission is accepted**.
- Answers Release: October 19.
- Submission in PDF (only) via canvas (emails with attachments will be ignored).
- Only typed PDF documents are accepted, no handwritten scanned documents.

1 Bayes' Rule

Q1 Probabilities Computation (4*5 points)

Suppose you are given a chance to win bonus grade points:
There are three boxes. Only one box contains a special prize that will grant you 1 bonus points. After you have chosen a box B_1 (B_1 is kept closed), one of the two remaining boxes will be opened (called B_2) such that it must not contain the prize (note that there is at least one such box).

(i) What is the probability of B_1 contains prize, $P(B_1 = 1)$?

$$P(B_1 = 1) = \frac{1}{3}$$

(ii) What is the likelihood probability of B_2 does not contains prize if B_1 contains prize, $P(B_2 = 0|B_1 = 1)$?

$$P(B_2 = 0|B_1 = 1) = 1$$

(iii) What is the probability of B_1 contains prize given B_2 does not contain prize, $P(B_1 = 1|B_2 = 0)$?

$$P(B_1 = 1|B_2 = 0) = \frac{P(B_2=0|B_1=1)P(B_1=1)}{P(B_2=0)} = \frac{1\times\frac{1}{3}}{1} = \frac{1}{3}$$

(iv) Now you are are given a second chance to choose boxes. You can either stick to B_1 or choose the only left box B_3. According to the Bayes decision rule, should you change your choice or not?

$$P(B_3 = 1|B_2 = 0) = P(B_3 = 0|B_2 = 0) = 1 - \tfrac{1}{3} = \tfrac{2}{3}$$

So my best choice is to choose B_3.

2 Naïve Bayes Classifier

Q2 Building a Naïve Bayes Classifier for Text (4*10 points)

Suppose you want to build a Naïve Bayes classifier for text. First of all, you represent a text document as if it were a bag-of-words. Naïve Bayes is a probabilistic classifier, meaning that for a document d, out of all classes $k \in \{1, 2, \cdots, K\}$ the classifier returns the class $\hat{y}$ which has the maximum probability given the document.

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} P(y = k|d) \tag{1}$$

This idea of Bayesian inference has been known since the work of Bayes (1763), and was first applied to text classification by Mosteller and Wallace (1964). The intuition of Bayesian classification is to use Bayes' rule to transform Eq.(1) into other probabilities that have some useful properties. Bayes' rule is presented in Eq.(2); it gives us a way to break down any conditional probability $P(x|y)$ into three other probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{2}$$

We can then substitute Eq.(2) into Eq.(1) to get Eq.(3):

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} P(y = k|d) = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} \frac{P(d|y = k)P(y = k)}{P(d)} \tag{3}$$

Because $P(d)$ doesn't change for each class, we can choose the class that maximizes this simpler formula:

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} P(y = k|d) = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} P(d|y = k)P(y = k) \tag{4}$$

We thus compute the most probable class $\hat{y}$ given some document d by choosing the class which has the highest product of two probabilities: the probability of the class $P(y = k)$ and the likelihood of the document $P(d|y = k)$:

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} \underbrace{P(d|y = k)}_{\text{likelihood}} \overbrace{P(y = k)}^{\text{probability of class } k} \tag{5}$$

To apply the Naïve Bayes classifier to text, we need to consider how to represent documents. Bag of words is an easy method to model documents. And it assumes the position doesn't matter. Thus we can consider word positions, by simplywalking an index through every word position in the document:

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} P(y=k) \prod_{i\in positions} P(w_i|y=k) \tag{6}$$

Naïve Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq.(6) is generally instead expressed as

$$\hat{y} = \underset{k\in\{1,2,\cdots,K\}}{\operatorname{argmax}} \log P(y=k) + \sum_{i\in positions} \log P(w_i|y=k) \tag{7}$$

By considering features in log space Eq.(7) computes the predicted class as a linear function of input features (here are words in the document). Classifiers that use a linear combination of the inputs to make a classification decision are called linear classifiers.

(i) Let N_k be the number of documents in our training data with class k and N_{doc} be the total number of documents. What is the maximum likelihood estimate of the probability of each class $P(y=k)$?

$$P(y=k) = \frac{N_k}{N_{doc}}$$

(ii) We use bag-of-words representation. We first concatenate all documents with class k into one big text. Let V be the union of all the words in all classes, w_i be the i-th word in the document, and $count(w_i, k)$ be the frequency of w_i. Then what is the maximum likelihood estimate of the probability $P(w_i|y=k)$?

$$P(w_i|y=k) = \frac{count(w_i,k)}{\sum_{w\in V} count(w,k)}$$

(iii) But suppose there are no training documents that both contain the word "fantastic" and are classified as class k. In such a case the probability for this feature will be zero. But since Naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero. What will you do to avoid this situation? Please write down your solution and improved formula.
Use Laplace Smoothing to avoid zero probability.

$$P(w_i|y=k) = \frac{count(w_i,k)+1}{\sum_{w\in V} count(w,k)+|V|}$$

(iv) Because some words did not occur in any training document in any class, they are not in our vocabulary V. What should you do for such unknown words? And very frequent words like *the* and *a* and number are unhelpful for prediction, what should you do for such words?
For unknown words, we can simply ignore them—remove them from test documents and not include any probability for them at all.
For high-frequency words, or called stop words, we can also remove them from all documents.

Table 1: training and test documents

	Class	Documents
Training	$+$	great
	$-$	very annoying
	$+$	the best one
	$-$	this version is rubbish
	$+$	awesome
	$+$	this version is easy to use
	$+$	good
	$-$	so terrible
Test	?	very easy and great

Q3 Running the Classifier on an Example (4*10 points)

According to what you have done in Q2, we will use a sentiment analysis domain with the two classes positive (+) and negative (−), and take the following miniature training and test documents. You need to compute:

(i) the probability of each class, $P(y = +)$ and $P(y = -)$

$$P(y = +) = \frac{N_+}{N_{doc}} = \frac{5}{8} = 0.625$$
$$P(y = -) = \frac{N_-}{N_{doc}} = \frac{3}{8} = 0.375$$

(ii) the probabilities for every word in the test document, $P(w_i|y = +)$ and $P(w_i|y = -)$

$$P(very|y = +) = 0.000$$
$$P(very|y = -) = 0.125$$
$$P(easy|y = +) = 0.083$$
$$P(easy|y = -) = 0.000$$
$$P(great|y = +) = 0.083$$
$$P(great|y = -) = 0.000$$

(iii) apply the strategy in Q2 (iii) to i and ii

$$P(y = +) = \frac{N_+}{N_{doc}} = \frac{5+1}{8+2} = 0.600$$
$$P(y = -) = \frac{N_-}{N_{doc}} = \frac{3+1}{8+2} = 0.400$$
$$P(very|y = +) = 0.034$$
$$P(very|y = -) = 0.080$$
$$P(easy|y = +) = 0.069$$
$$P(easy|y = -) = 0.040$$
$$P(great|y = +) = 0.069$$
$$P(great|y = -) = 0.040$$

(iv) the probabilities for each class, $P(y = +|d)$ and $P(y = -|d)$, and your final decision, positive or negative

$$P(y = +|d) = 0.658$$
$$P(y = -|d) = 0.342$$

Thus, this test document is positive.

You are required to solve these questions with the help of code in Lab 4. Then you can compute by hand to check the results.
You don't need to handle very frequent words!

Appendix A

To help you understand Lab4 code, we have designed some unit tests.

Normalization without Laplace Smoothing

```
def normalize(P):
    # without Laplace smoothing
    norm = np.sum(P, axis=0, keepdims=False)
    return P / norm
```

```
Input1:
    P = np.array([1,2,3,4])
Output1:
    np.array([0.1, 0.2, 0.3, 0.4])

Input2:
    P = np.array([[1, 2, 3, 4],
                  [4, 3, 2, 1]])
Output2:
    np.array([[0.2, 0.4, 0.6, 0.8],
              [0.8, 0.6, 0.4, 0.2]])
```

Normalization with Laplace Smoothing

```
def normalize(P):
    # with Laplace smoothing
    K = P.shape[0]
    norm = np.sum(P, axis=0, keepdims=True)
    return (P + 1) / (norm + K)
```

```
Input1:
    P = np.array([1,2,3,4])
Output1:
    P = np.array([0.14285714, 0.21428571, 0.28571429, 0.35714286])

Input2:
    P = np.array([[1, 2, 3, 4],
                  [4, 3, 2, 1]])
Output2:
    P = np.array([[0.28571429, 0.42857143, 0.57142857, 0.71428571],
                  [0.71428571, 0.57142857, 0.42857143, 0.28571429]])
```

$P(y)$ Calculation

```
# normalization with laplace smoothing
P_y = normalize(np.sum(data_delta, axis=0, keepdims=False))
```

```
Input1:
    data_delta = np.array([[1,0],
                           [1,0],
                           [0,1],
                           [1,0]])
Output1:
    P_y = np.array([0.66666667, 0.33333333])

Input2:
    data_delta = np.array([[1,0],
                           [1,0],
                           [1,0],
                           [1,0]])
Output2:
    P_y = np.array([0.83333333, 0.16666667])
```

$P(w|y)$ Calculation

```
# normalization with laplace smoothing
# to make understand easily, use np.array to represent data_matrix
P_xy = normalize(data_matrix.transpose().dot(data_delta))
```

```
Input1:
    data_matrix = np.array([[0,1,0,0,0],
                            [1,0,1,0,0],
                            [0,1,0,0,1],
                            [0,0,0,1,0]])
    data_delta = np.array([[1,0],
                           [1,0],
                           [0,1],
                           [1,0]])
Output1:
    P_xy = np.array([[0.22222222, 0.14285714],
                     [0.22222222, 0.28571429],
                     [0.22222222, 0.14285714],
                     [0.22222222, 0.14285714],
                     [0.11111111, 0.28571429]])

Input2:
    data_matrix = np.array([[0,1,0,0,0],
                            [1,0,1,0,0],
                            [0,1,0,0,1],
                            [0,0,0,1,0]])
    data_delta = np.array([[1,0],
                           [1,0],
                           [1,0],
                           [1,0]])
Output2:
    P_xy = np.array([[0.18181818, 0.2],
                     [0.27272727, 0.2],
                     [0.18181818, 0.2],
                     [0.18181818, 0.2],
                     [0.18181818, 0.2]])
```

$\log(P(y)P(d|y))$ Calculation

```
log_P_y = np.expand_dims(np.log(P_y), axis=0) # N * K
log_P_xy = np.log(P_xy)
log_P_dy = data_matrix.dot(log_P_xy)
log_P = log_P_y + log_P_dy

Input1:
    P_y = np.array([0.66666667, 0.33333333])
    P_xy = np.array([[0.22222222, 0.14285714],
                     [0.22222222, 0.28571429],
                     [0.22222222, 0.14285714],
                     [0.22222222, 0.14285714],
                     [0.11111111, 0.28571429]])
    data_matrix = np.array([[0,1,0,0,0],
                            [1,0,1,0,0],
                            [0,1,0,0,1],
                            [0,0,0,1,0]])
Output1:
    log_P = np.array([[-1.90954251, -2.35137525],
                      [-3.41361992, -4.99043264],
                      [-4.1067671 , -3.60413821],
                      [-1.90954251, -3.04452247]])
Input2:
    P_y = np.array([0.83333333, 0.16666667])
    P_xy = np.array([[0.18181818, 0.2],
                     [0.27272727, 0.2],
                     [0.18181818, 0.2],
                     [0.18181818, 0.2],
                     [0.18181818, 0.2]])
    data_matrix = np.array([[0,1,0,0,0],
                            [1,0,1,0,0],
                            [0,1,0,0,1],
                            [0,0,0,1,0]])
Output2:
    log_P = np.array([[-1.48160455, -3.40119736],
                      [-3.59181777, -5.01063527],
                      [-3.18635266, -5.01063527],
                      [-1.88706966, -3.40119736]])
```

Appendix B

To finish Q3 with the help of Lab4 code, we need change training data and test data firstly, where label=-1 means the test example, label=1 means the negative example and label=2 means the positive example.

Table 2: training data and test data

data/train.csv	data/test.csv
id,text,label	id,text,label
0,great,2	0,very easy and great,-1
1,very annoying,1	
2,the best one,2	
3,this version is rubbish,1	
4,awesome,2	
5,this version is easy to use,2	
6,good,2	
7,so terrible,1	

Because we don't need to handle frequent words, aka stop words. We can simply modify the *stop_words*:
change

```
stop_words = set(stopwords.words('english') + list(string.punctuation))
```

to

```
stop_words = set()
```

To finish Q3(i) and Q3(ii) without Laplace smoothing, we also need to modify the *normalize* function:
change

```
def normalize(P):
    K = P.shape[0]
    norm = np.sum(P, axis=0, keepdims=True)
    return (P + 1) / (norm + K)
```

to

```
def normalize(P):
    K = P.shape[0]
    norm = np.sum(P, axis=0, keepdims=True)
    return P / norm
```

In Q3(i) and Q3(ii), we need to compute $P(y = +)$, $P(y = -)$ and $P(w_i|y = +)$, $P(w_i|y = -)$, which have been calculated in Lab4. And they correspond to

```
P_y, P_xy = train_NB(train_data_label, train_data_matrix)
```

We can use print to get results.

```
print("P_y")
print(P_y)
print("P_xy")
for i, word in enumerate(vocab):
    print(word, '␣' * (10-len(word)), P_xy[i])
```

After running the code, we will get

```
P_y
[0.375 0.625]
P_xy
this        [0.125      0.08333333]
rubbish     [0.125  0.          ]
is          [0.125      0.08333333]
great       [0.         0.08333333]
annoying    [0.125  0.          ]
the         [0.         0.08333333]
best        [0.         0.08333333]
one         [0.         0.08333333]
easy        [0.         0.08333333]
good        [0.         0.08333333]
so          [0.125  0.          ]
terrible    [0.125  0.          ]
to          [0.         0.08333333]
awesome     [0.         0.08333333]
very        [0.125  0.          ]
version     [0.125      0.08333333]
use         [0.         0.08333333]
```

Therefore,

$$P(y=+) = 0.625$$
$$P(y=-) = 0.375$$
$$P(very|y=+) = 0.000$$
$$P(very|y=-) = 0.125$$
$$P(easy|y=+) = 0.083$$
$$P(easy|y=-) = 0.000$$
$$P(great|y=+) = 0.083$$
$$P(great|y=-) = 0.000$$

Then we use Laplace smoothing when normalizing:

```
def normalize(P):
    K = P.shape[0]
    norm = np.sum(P, axis=0, keepdims=True)
    return (P + 1) / (norm + K)
```

And we can get different results:

```
P_y
[0.4  0.6]
P_xy
very          [0.08        0.03448276]
is            [0.08        0.06896552]
easy          [0.04        0.06896552]
great         [0.04        0.06896552]
good          [0.04        0.06896552]
so            [0.08        0.03448276]
use           [0.04        0.06896552]
rubbish       [0.08        0.03448276]
terrible      [0.08        0.03448276]
the           [0.04        0.06896552]
one           [0.04        0.06896552]
version       [0.08        0.06896552]
best          [0.04        0.06896552]
annoying      [0.08        0.03448276]
to            [0.04        0.06896552]
this          [0.08        0.06896552]
awesome       [0.04        0.06896552]
```

Now, the answer of Q3(iii) is:

$$\begin{gathered}
P(y=+) = 0.600\\
P(y=-) = 0.400\\
P(very|y=+) = 0.034\\
P(very|y=-) = 0.080\\
P(easy|y=+) = 0.069\\
P(easy|y=-) = 0.040\\
P(great|y=+) = 0.069\\
P(great|y=-) = 0.040
\end{gathered}$$

As we can see, Laplace smoothing avoids zero-probability.
Some students are confused about why applying Laplace smoothing to $P(y=+)$ and $P(y=-)$. Will it affect the performance? In a large dataset, applying Laplace smoothing on $P(y)$ will not affect your model, because the denominator (the number of documents) is much larger than the number of classes. Some people use smoothing on $P(y)$ (e.g. Eq.(3.6) NigamEtAl-bookChapter) and others do not. Lab4 code follows the former. Just image that if we have an empty document, we can random guess a class or guess according to the class distribution. Random guessing strategy is the "1" we add to every class.

Finally, we need to compute $P(y=+|d)$ and $P(y=-|d)$ in Q3(iv). According to Eq.(2), we know that:

$$P(y=+|d) = \frac{P(d|y=+)P(y=+)}{P(d)} \tag{8}$$

$$P(y=-|d) = \frac{P(d|y=-)P(y=-)}{P(d)} \tag{9}$$

We have got $P(y=+)$ and $P(y=-)$ in Q3(iii), and we can get $P(d|y=+)$ according to (10), and $P(d|y=-)$ in the similar way.

$$\begin{aligned} P(d|y=+) &= \prod_{i \in positions} P(w_i|y=+) \\ &= \exp\,(\sum_{i \in positions} \log P(w_i|y=+)) \end{aligned} \tag{10}$$

$$\begin{aligned} P(d|y=-) &= \prod_{i \in positions} P(w_i|y=-) \\ &= \exp\,(\sum_{i \in positions} \log P(w_i|y=-)) \end{aligned} \tag{11}$$

What remains to do is to compute $P(d)$. Because we know $P(y=+|d) + P(y=-|d) = 1$, then

$$\begin{aligned} P(y=+|d) + P(y=-|d) &= \frac{P(d|y=+)P(y=+)}{P(d)} + \frac{P(d|y=-)P(y=-)}{P(d)} \\ &= \frac{P(d|y=+)P(y=+) + P(d|y=-)P(y=-)}{P(d)} \\ &= 1 \end{aligned} \tag{12}$$

Therefore, we just need to normalize $P(d|y=+)P(y=+)$ and $P(d|y=-)P(y=-)$.

In Lab4 code, we use code following to compute $\log(P(d|y=+)P(y=+))$ and $\log(P(d|y=-)P(y=-))$.

```
log_P_xy = np.log(P_xy)
log_P_dy = data_matrix.dot(log_P_xy)
log_P = log_P_y + log_P_dy
```

Two lines of code can satisfy our demand:

```
P = np.exp(log_P)
print(P / np.sum(P, axis=-1, keepdims=True))
```

The first line is to compute $P(d)$, and the second line is to normalize probabilities. Finally, we can get the answer:

$$P(y=+|d) = 0.658$$
$$P(y=-|d) = 0.342$$

Thus, this test document is positive.