CS440/ECE448 Spring 2023

MP02: Naive Bayes

The first thing you need to do is to download mp02.zip. Content is similar to MP01.

This file (`mp02_notebook.ipynb`) will walk you through the whole MP, giving you instructions and debugging tips as you go.

Table of Contents

Reading the data

The dataset in your template package consists of 10000 positive and 3000 negative movie reviews. It is a subset of the Stanford Movie Review Dataset, which was originally introduced by this paper. We have split this data set for you into 5000 development examples and 8000 training examples. The autograder also has a hidden set of test examples, generally similar to the development dataset.

The data folder is structured like this:

```
├─ train
│   ├─ neg
│   │   └─ 2000 negative movie reviews (text)
│   └─ pos
│       └─ 6000 positive movie reviews (text)
└─ dev
    ├─ neg
    │   └─ 1000 negative movie reviews (text)
    └─ pos
│       └─ 4000 positive movie reviews (text)
```

In order to help you load the data, we provide you with a utility function called `reader.py` . This has two new functions that didn't exist in mp01:

- loadTrain: load a training set
- loadDev: load a dev set

In [1]:
```
import reader, importlib
importlib.reload(reader)
help(reader.loadTrain)
```

```
Help on function loadTrain in module reader:

loadTrain(dirname, stemming, lower_case, use_tqdm=True)
    Loads a training dataset.

    Parameters:
    dirname (str): the directory containing the data
        - dirname/y should contain training examples from class y

    stemming (bool): if True, use NLTK's stemmer to remove suffixes
    lower_case (bool): if True, convert letters to lowercase
    use_tqdm (bool, default:True): if True, use tqdm to show status bar

    Output:
    train (dict of list of lists):
        - train[y][i][k] = k'th token of i'th text of class y
```

This time the text files have not been lowercased for you in advance, so you probably want to lowercase them using the `lower_case` bool:

In [2]:
```
importlib.reload(reader)

train = reader.loadTrain('data/train', False, True)
```

```
100%|██████████| 2000/2000 [00:00<00:00, 10929.16it/s]
100%|██████████| 6000/6000 [00:00<00:00, 11735.77it/s]
```

In [3]:
```
for y in train.keys():
    print("There were",len(train[y]),"texts loaded for class",y)
```

```
There were 2000 texts loaded for class neg
There were 6000 texts loaded for class pos
```

In [4]:
```
print("The first positive review is:",train['pos'][0])
```

```
The first positive review is: ['i', 'went', 'and', 'saw', 'this', 'movie',
'last', 'night', 'after', 'being', 'coaxed', 'to', 'by', 'a', 'few', 'frien
ds', 'of', 'mine', 'i', 'll', 'admit', 'that', 'i', 'was', 'reluctant', 't
o', 'see', 'it', 'because', 'from', 'what', 'i', 'knew', 'of', 'ashton', 'k
utcher', 'he', 'was', 'only', 'able', 'to', 'do', 'comedy', 'i', 'was', 'wr
ong', 'kutcher', 'played', 'the', 'character', 'of', 'jake', 'fischer', 've
ry', 'well', 'and', 'kevin', 'costner', 'played', 'ben', 'randall', 'with',
'such', 'professionalism', 'the', 'sign', 'of', 'a', 'good', 'movie', 'is',
'that', 'it', 'can', 'toy', 'with', 'our', 'emotions', 'this', 'one', 'di
d', 'exactly', 'that', 'the', 'entire', 'theater', 'which', 'was', 'sold',
'out', 'was', 'overcome', 'by', 'laughter', 'during', 'the', 'first', 'hal
f', 'of', 'the', 'movie', 'and', 'were', 'moved', 'to', 'tears', 'during',
'the', 'second', 'half', 'while', 'exiting', 'the', 'theater', 'i', 'not',
'only', 'saw', 'many', 'women', 'in', 'tears', 'but', 'many', 'full', 'grow
n', 'men', 'as', 'well', 'trying', 'desperately', 'not', 'to', 'let', 'anyo
ne', 'see', 'them', 'crying', 'this', 'movie', 'was', 'great', 'and', 'i',
'suggest', 'that', 'you', 'go', 'see', 'it', 'before', 'you', 'judge']
```

Learning a Naive Bayes Model: Maximum Likelihood

In order to understand Naive Bayes, it might be useful to know the difference between bigram types and bigram tokens.

- **Bigram token:** A bigram token consists of two consecutive word tokens from the text. For grading purpose, all bigrams are represented as word1*-*-*-*word2 , i.e. *-*-*-* as a separator, and not as a tuple. In the context of bigrams, the tokens are these pairs of words. To emphasize, note that our definition of bigram is two "consecutive" word tokens. Consecutive!
- **Bigram type:** Bigram types are the set of unique pairs of consecutive words that occurred in a text. The number of bigram types in the n^{th} positive text can be found by first generating all bigram tokens from the text and then counting the unique bigrams.

A Naive Bayes model consists of two types of probability distributions:

- The **prior** is the distribution over classes, $P(\text{Class})$.
- The **likelihood** is the probability of a bigram token given a particular class, $P(\text{Bigram}|\text{Class})$.

The prior can be estimated from the training data. In your training data, $P(\text{Class} = \text{pos}) = 0.75$.

Often, though, the testing data will have a different class distribution than the training data. If you don't know the testing priors, then it's sometimes best to just assume a uniform distribution, i.e., $P(\text{Class} = \text{pos}) = 0.5$.

The likelihood is the informative part of a Naive Bayes model: it tells you which words are used more often in negative versus positive movie reviews.

There are many ways in which you can estimate the likelihood. The following formula is called the **maximum likelihood** estimate, because it maximizes the likelihood of the words in your training dataset:

$$P(\text{Bigram} = x|\text{Class} = y) = \frac{\backslash\#\text{ tokens of bigram } x \text{ in texts of class } y}{\backslash\#\text{ tokens of any bigram in texts of class } y}$$

In this part of the MP, you will estimate what are called **frequency tables**. The frequency of x given y is the number of times that bigram x occurred in texts of class y. The relevant method in `submitted.py` is the one called `create_frequency_table`:

In [5]:
```
import submitted, importlib
importlib.reload(submitted)
help(submitted.create_frequency_table)
```

```
Help on function create_frequency_table in module submitted:

create_frequency_table(train)
    Parameters:
    train (dict of list of lists)
        - train[y][i][k] = k'th token of i'th text of class y

    Output:
    frequency (dict of Counters):
        - frequency[y][x] = number of occurrences of bigram x in texts of c
lass y,
          where x is in the format 'word1*-*-*-*word2'
```

Edit `create_frequency_table` so that it does what its docstring says it should do.

Hint: your code will be shorter if you use the python data structure called a Counter.

When your code works, you should get the following results:

In [6]:
```
importlib.reload(submitted)
frequency = submitted.create_frequency_table(train)

print("frequency['pos'][('this', 'film')]=",frequency['pos']['this*-*-*-*fi
print("frequency['neg'][('this', 'film')]=",frequency['neg']['this*-*-*-*fi
print("\n")

print("frequency['pos'][('the', 'movie')]=",frequency['pos']['the*-*-*-*mov
print("frequency['neg'][('the', 'movie')]=",frequency['neg'][('the*-*-*-*mo
print("\n")


print("frequency['pos'][('of', 'the')]=",frequency['pos']['of*-*-*-*the'])
print("frequency['neg'][('of', 'the')]=",frequency['neg']['of*-*-*-*the'])
print("\n")

print("frequency['pos'][('to', 'be')]=",frequency['pos']['to*-*-*-*be'])
print("frequency['neg'][('to', 'be')]=",frequency['neg']['to*-*-*-*be'])
print("\n")

print("frequency['pos'][('and', 'the')]=",frequency['pos']['and*-*-*-*the']
print("frequency['neg'][('and', 'the')]=",frequency['neg']['and*-*-*-*the']
print("\n")

print("--------------------------------------\n")

print("Total # tokens in pos texts is",sum(frequency['pos'].values()))
print("Total # tokens in neg texts is",sum(frequency['neg'].values()))
print("\n")

print("Total # types in pos texts is",len(frequency['pos'].keys()))
print("Total # types in neg texts is",len(frequency['neg'].keys()))
```

```
frequency['pos'][('this', 'film')]= 2656
frequency['neg'][('this', 'film')]= 879


frequency['pos'][('the', 'movie')]= 2507
frequency['neg'][('the', 'movie')]= 1098


frequency['pos'][('of', 'the')]= 10008
frequency['neg'][('of', 'the')]= 2779


frequency['pos'][('to', 'be')]= 2500
frequency['neg'][('to', 'be')]= 1057


frequency['pos'][('and', 'the')]= 3342
frequency['neg'][('and', 'the')]= 983


--------------------------------------

Total # tokens in pos texts is 1421513
Total # tokens in neg texts is 468194


Total # types in pos texts is 475651
Total # types in neg texts is 195021
```

Learning a Naive Bayes model: Stop words

There are many common word pairs (bigrams) that may seem to be unrelated to whether a movie review is positive or negative. Due to the nature of the training data, it's possible that some bigrams, consisting entirely of common words like "is", "of", "and", etc., are more frequent in one part of the training data than another. This can be problematic, as it means a test review might be wrongly classified as "positive" just because it contains many instances of innocuous bigrams like ("and", "the").

A "stopword list" is a list of words that should not be considered when you classify a test text. In the context of bigrams, we consider a bigram as a stopword if both of its constituent words are in the stopword list. There are many candidate stopword lists available on the internet. The stopword list that we've provided for you is based on this one: https://www.ranks.nl/stopwords.

To emphasize, we consider a bigram as a stopword if "both" of its constituent words are in the stopword list. Both!

Here is our stopword list:

In [7]:
```
importlib.reload(submitted)
print(sorted(submitted.stopwords))
```

```
["'d", "'ll", "'m", "'re", "'s", "'t", "'ve", 'a', 'about', 'above', 'afte
r', 'again', 'against', 'all', 'am', 'an', 'and', 'any', 'are', 'aren', 'a
s', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between',
'both', 'but', 'by', 'can', 'cannot', 'could', 'couldn', 'did', 'didn', 'd
o', 'does', 'doesn', 'doing', 'don', 'down', 'during', 'each', 'few', 'fo
r', 'from', 'further', 'had', 'hadn', 'has', 'hasn', 'have', 'haven', 'havi
ng', 'he', 'her', 'here', 'hers', 'herself', 'him', 'himself', 'his', 'ho
w', 'i', 'if', 'in', 'into', 'is', 'isn', 'it', 'its', 'itself', 'let', 'l
l', 'me', 'more', 'most', 'mustn', 'my', 'myself', 'no', 'nor', 'not', 'o
f', 'off', 'on', 'once', 'only', 'or', 'other', 'ought', 'our', 'ours', 'ou
rselves', 'out', 'over', 'own', 'same', 'shan', 'she', 'should', 'shouldn',
'so', 'some', 'such', 'than', 'that', 'the', 'their', 'theirs', 'them', 'th
emselves', 'then', 'there', 'these', 'they', 'this', 'those', 'through', 't
o', 'too', 'under', 'until', 'up', 'very', 'was', 'wasn', 'we', 'were', 'we
ren', 'what', 'when', 'where', 'which', 'while', 'who', 'whom', 'why', 'wit
h', 'won', 'would', 'wouldn', 'you', 'your', 'yours', 'yourself', 'yourselv
es']
```

To effectively avoid counting bigrams that are considered stopwords, two steps are necessary:

1. Pretend that the frequency of bigrams, where both words are stopwords, in the training corpus is zero.
2. Ignore such bigrams if they occur in testing data.

In this part of the MP, you should set the frequencies of those bigram stopwords to zero. Use the `del` command (see Counters), so that these bigrams don't get counted among either the bigram types or the bigram tokens.

In [8]:
```
importlib.reload(submitted)
help(submitted.remove_stopwords)
```

```
Help on function remove_stopwords in module submitted:

remove_stopwords(frequency)
    Parameters:
    frequency (dict of Counters):
        - frequency[y][x] = number of occurrences of bigram x in texts of c
lass y,
          where x is in the format 'word1*-*-*-*word2'
    stopwords (set of str):
        - Set of stopwords to be excluded

    Output:
    nonstop (dict of Counters):
        - nonstop[y][x] = frequency of bigram x in texts of class y,
          but only if neither token in x is a stopword. x is in the format
'word1*-*-*-*word2'
```

In [9]:
```
importlib.reload(submitted)
nonstop = submitted.remove_stopwords(frequency)

print("frequency['pos'][('this', 'film')]=",frequency['pos']['this*-*-*-*fi
print("frequency['pos'][('this', 'film')]=",nonstop['pos']['this*-*-*-*film
print("\n")

print("frequency['pos'][('the', 'movie')]=",frequency['pos']['the*-*-*-*mov
print("frequency['pos'][('the', 'movie')]=",nonstop['pos']['the*-*-*-*movie
print("\n")
print("frequency['pos'][('of', 'the')]=",frequency['pos']['of*-*-*-*the'])
print("frequency['pos'][('of', 'the')]=",nonstop['pos']['of*-*-*-*the'])
print("\n")

print("frequency['pos'][('to', 'be')]=",frequency['pos']['to*-*-*-*be'])
print("frequency['pos'][('to', 'be')]=",nonstop['pos']['to*-*-*-*be'])
print("\n")

print("frequency['pos'][('and', 'the')]=",frequency['pos']['and*-*-*-*the']
print("frequency['pos'][('and', 'the')]=",nonstop['pos']['and*-*-*-*the'])
print("\n")

print("--------------------------------------\n")

print("Total pos frequency:",sum(frequency['pos'].values()))
print("Total pos non-stopwords",sum(nonstop['pos'].values()))
print("\n")

print("Total # types in pos texts is",len(frequency['pos'].keys()))
print("Total # non-stopwords in pos is",len(nonstop['pos'].keys()))

print("Length of the stopwords set is:",len(submitted.stopwords))
```

```
frequency['pos'][('this', 'film')]= 2656
frequency['pos'][('this', 'film')]= 2656


frequency['pos'][('the', 'movie')]= 2507
frequency['pos'][('the', 'movie')]= 2507


frequency['pos'][('of', 'the')]= 10008
frequency['pos'][('of', 'the')]= 0


frequency['pos'][('to', 'be')]= 2500
frequency['pos'][('to', 'be')]= 0


frequency['pos'][('and', 'the')]= 3342
frequency['pos'][('and', 'the')]= 0


--------------------------------------

Total pos frequency: 1421513
Total pos non-stopwords 1168682


Total # types in pos texts is 475651
Total # non-stopwords in pos is 468246
Length of the stopwords set is: 150
```

Learning a Naive Bayes model: Laplace Smoothing

The maximum likelihood formula results in some words having zero probability, just because they were not contained in your training data. A better formula is given by Laplace smoothing, according to which

$$P(\text{Bigram} = x|\text{Class} = y) = \frac{(\#\text{ tokens of bigram } x \text{ in texts of clas}}{(\#\text{ tokens of any bigram in texts of class } y) + k \times (\backslash}$$

...where k is a hyperparameter that is usually chosen by trying several different values, and choosing the value that gives you the best accuracy on your development dataset.

The `+1` in the denominator is used to account for bigrams that were never seen in the training dataset for class y. All such words are mapped to the type `OOV` (out of vocabulary), which has the likelihood

$$P(\text{Token}=\text{OOV}|\text{Class}=y)=\frac{k}{(\#\text{ tokens of any bigram in texts of class }y)+k\times$$

In this part of the MP, the method you'll create in `submitted.py` is called `laplace_smoothing`.

In [10]:
```
importlib.reload(submitted)
help(submitted.laplace_smoothing)
```

```
Help on function laplace_smoothing in module submitted:

laplace_smoothing(nonstop, smoothness)
    Parameters:
    nonstop (dict of Counters)
        - nonstop[y][x] = frequency of bigram x in y, where x is in the for
mat 'word1*-*-*-*word2'
          and neither word1 nor word2 is a stopword
    smoothness (float)
        - smoothness = Laplace smoothing hyperparameter

    Output:
    likelihood (dict of dicts)
        - likelihood[y][x] = Laplace-smoothed likelihood of bigram x given
y,
          where x is in the format 'word1*-*-*-*word2'
        - likelihood[y]['OOV'] = likelihood of an out-of-vocabulary bigram
given y


    Important:
    Be careful that your vocabulary only counts bigrams that occurred at le
ast once
    in the training data for class y.
```

In [11]:
```
importlib.reload(submitted)
likelihood = submitted.laplace_smoothing(frequency, 0.001)

print("likelihood['pos'][('this', 'film')]=",likelihood['pos']['this*-*-*-*
print("likelihood['neg'][('this', 'film')]=",likelihood['neg']['this*-*-*-*
print("\n")

print("likelihood['pos']['OOV']=",likelihood['pos']['OOV'])
print("likelihood['neg']['OOV']=",likelihood['neg']['OOV'])
print("\n")

print("(should be approx. 1): likelihood['pos'] sums to",sum(likelihood['pos
print("(should be approx. 1): Likelihood['neg'] sums to",sum(likelihood['neg
likelihood['pos'][('this', 'film')]= 0.0018678074513916727
likelihood['neg'][('this', 'film')]= 0.0018766473139073699


likelihood['pos']['OOV']= 7.032404925267997e-10
likelihood['neg']['OOV']= 2.134977450432218e-09


(should be approx. 1): likelihood['pos'] sums to 0.9999999999864526
(should be approx. 1): Likelihood['neg'] sums to 1.0000000000037939
```

Decisions using a Naive Bayes model

Suppose you are given a text, which is just a list of word tokens, $x = [x_1, \ldots, x_n]$. You want to decide whether this text is a positive movie review or a negative review. According to decision theory, the probability of error is minimized by the following rule:

$$\text{Estimated Class} = \begin{cases} \text{pos if } P(\text{Class} = \text{pos}|\text{Text} = x) > P(\text{Class} = \text{neg}|\text{Text} = x) \\ \text{neg if } P(\text{Class} = \text{pos}|\text{Text} = x) < P(\text{Class} = \text{neg}|\text{Text} = x) \\ \text{undecided if } P(\text{Class} = \text{pos}|\text{Text} = x) = P(\text{Class} = \text{neg}|\text{Text} \end{cases}$$

For a bigram model, the text x is considered as a sequence of bigram tokens $[(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)]$. The posterior probabilities $P(\text{Class}|\text{Text})$ can be estimated using the Naive Bayes model:

$$P(\text{Class} = y|\text{Text} = x) = \frac{P(\text{Class} = y)}{P(\text{Text} = x)} \prod_{i \notin \text{stopword bigrams}, i=1}^{n-1} P(\text{Token} = x_i|\text{Class} = y$$

Implementation Details

Notice some details:

1. The term $P(\text{Text} = x)$ doesn't depend on y. If you're trying to figure out which is bigger, $P(\text{pos}|x)$ or $P(\text{neg}|x)$, then you don't need to calculate it.
2. Multiplying together n probabilities will result in a number that your computer might round down to 0. In order to prevent that, take the logarithm of both sides of the equation above.
3. If x_i is a stopword bigram, don't calculate its likelihood. If it isn't a stopword bigram, but it doesn't have an entry in `likelihood[y]`, then you should use `likelihood[y]["OOV"]`.

Implementation

For this part of the MP, finish the method called `submitted.naive_bayes`:

In [12]:
```
importlib.reload(submitted)
help(submitted.naive_bayes)
```

```
Help on function naive_bayes in module submitted:

naive_bayes(texts, likelihood, prior)
    Parameters:
    texts (list of lists) -
        - texts[i][k] = k'th token of i'th text
    likelihood (dict of dicts)
        - likelihood[y][x] = Laplace-smoothed likelihood of bigram x given
y,
          where x is in the format 'word1*-*-*-*word2'
    prior (float)
        - prior = the prior probability of the class called "pos"

    Output:
    hypotheses (list)
        - hypotheses[i] = class label for the i'th text
```

Use `reader.loadDev` to load the dev set, then try classifying it with, say, a prior of 0.5:

In [13]:
```python
importlib.reload(reader)
texts, labels = reader.loadDev('data/dev', False, True, True)

for y in ['neg','pos']:
    print("There are",labels.count(y),'examples of class',y)
```

```
100%|██████████| 1000/1000 [00:00<00:00, 10638.69it/s]
100%|██████████| 4000/4000 [00:00<00:00, 11661.83it/s]
There are 1000 examples of class neg
There are 4000 examples of class pos
```

In [14]:
```python
importlib.reload(submitted)
hypotheses = submitted.naive_bayes(texts, likelihood, 0.5)

for y in ['neg','pos', 'undecided']:
    print("There are",hypotheses.count(y),'examples that were labeled with 
```

```
There are 808 examples that were labeled with class neg
There are 4192 examples that were labeled with class pos
There are 0 examples that were labeled with class undecided
```

In [15]:
```python
print(len(hypotheses))
print(len(labels))
```

```
5000
5000
```

In [16]:
```python
print("The accuracy of the classifier on the dev set is:")

count_correct = 0
for (y,yhat) in zip(labels, hypotheses):
    if y==yhat:
        count_correct += 1

print(count_correct / len(labels))
```

```
The accuracy of the classifier on the dev set is:
0.884
```

Optimizing Hyperparameters

The performance of the model is heavily influenced by two parameters that can't be measured from the training data:

1. The prior, $P(\text{Class} = \text{pos})$. The training and testing data might have different priors, so estimating this from the training data is suboptimal.
2. The Laplace smoothing parameter, k.

Since these two parameters can't be (correctly) estimated from the training data, they are called **hyperparameters**. Hyperparameters are usually determined based on your knowledge about the problem, or by running a lot of experiments to see which values give the best result on the development test data.

The function you'll write in this part of the MP is called `optimize_hyperparameters`.

In [17]:
```
importlib.reload(submitted)
help(submitted.optimize_hyperparameters)
```

```
Help on function optimize_hyperparameters in module submitted:

optimize_hyperparameters(texts, labels, nonstop, priors, smoothnesses)
    Parameters:
    texts (list of lists) - dev set texts
        - texts[i][k] = k'th token of i'th text
    labels (list) - dev set labels
        - labels[i] = class label of i'th text
    nonstop (dict of Counters)
        - nonstop[y][x] = frequency of word x in class y, x not stopword
    priors (list)
        - a list of different possible values of the prior
    smoothnesses (list)
        - a list of different possible values of the smoothness

    Output:
    accuracies (numpy array, shape = len(priors) x len(smoothnesses))
        - accuracies[m,n] = dev set accuracy achieved using the
          m'th candidate prior and the n'th candidate smoothness
```

Let's use this function to test some different candidate values for the prior and the smoothness. The values we test are a little arbitrary, but let's try the following:

In [18]:
```
importlib.reload(submitted)
import numpy as np

priors = [0.5,0.65,0.75]
smoothnesses = [0.0001,0.001,0.01]
accuracies = submitted.optimize_hyperparameters(texts,labels,nonstop,priors,

(m,n) = np.unravel_index(np.argmax(accuracies), accuracies.shape)
print("The best accuracy achieved was",accuracies[m,n])
print("It was achieved for a prior of", priors[m], "and a smoothness of", sı
```

```
The best accuracy achieved was 0.885
It was achieved for a prior of 0.75 and a smoothness of 0.001
```

Grade your homework

If you've reached this point, and all of the above sections work, then you're ready to try grading your homework! Before you submit it to Gradescope, try grading it on your own machine. This will run some visible test cases (which you can read in `tests/test_visible.py`), and compare the results to the solutions (which you can read in `solution.json`).

The exclamation point (!) tells python to run the following as a shell command. Obviously you don't need to run the code this way -- this usage is here just to remind you that you can also, if you wish, run this command in a terminal window.

In [20]:
```
!python grade.py
```

```
..........
----------------------------------------------------------------------
Ran 10 tests in 31.553s

OK
```

If you got any 'E' marks, it means that your code generated some runtime errors, and you need to debug those.

If you got any 'F' marks, it means that your code ran without errors, but that it generated results that are different from the solutions in `solutions.json`. Try debugging those differences.

If neither of those things happened, and your result was a series of dots, then your code works perfectly.

If you're not sure, you can try running grade.py with the -j option. This will produce a JSON results file, in which the best score you can get is 50.

Now you should try uploading `submitted.py` to Gradescope.

Gradescope will run the same visible tests that you just ran on your own machine, plus some additional hidden tests. It's possible that your code passes all the visible tests, but fails the hidden tests. If that happens, then it probably means that you hard-coded a number into your function definition, instead of using the input parameter that you were supposed to use. Debug by running your function with a variety of different input parameters, and see if you can get it to respond correctly in all cases.

Once your code works perfectly on Gradescope, with no errors, then you are done with the MP. Congratulations!