

Problem set 1, Intro to NLP 2018

This is due on September 25, 2018, submitted electronically. 100 points total.

How to do this problem set:

- What version of Python should I use? 3.6!
- Most of these questions require writing Python code and computing results, and the rest of them have textual answers. To generate the answers, you will have to fill out a supporting file, `hw1.py`.
- For all of the textual answers you have to fill out have placeholder text which says "Answer in one or two sentences here." For each question, you need to replace "Answer in one or two sentences here" with your answer.
- Write all the answers in this ipython notebook. Once you are finished (1) Generate a PDF via (File -> Download As -> PDF) and upload to Gradescope (2) Turn in `hw_1.py` and `hw_1.ipynb` on Moodle.
- **Important** check your PDF before you turn it in to gradescope to make sure it exported correctly. If ipython notebook gets confused about your syntax it will sometimes terminate the PDF creation routine early. If your whole PDF does not print, try running `$jupyter nbconvert --to pdf 2018hw1.ipynb` to identify and fix any syntax errors that might be causing problems
- When creating your final version of the PDF to hand in, please do a fresh restart and execute every cell in order. Then you'll be sure it's actually right. One handy way to do this is by clicking `Cell -> Run All` in the notebook menu.
- This assignment is designed so that you can run all cells in a few minutes of computation time. If it is taking longer than that, you probably have made a mistake in your code.

Academic honesty

- We will audit the Moodle code from a set number of students, chosen at random. The audits will check that the code you wrote and turned on Moodle generates the answers you turn in on your PDF. If you turn in correct answers on your PDF without code that actually generates those answers, we will consider this a serious case of cheating. See the course page for honesty policies.
- We will also run automatic checks of code on Moodle for plagiarism. Copying code from others is also considered a serious case of cheating.

```
In [ ]: # Run this cell! It sets some things up for you.

# This code makes plots appear inline in this document rather than in a new
import matplotlib.pyplot as plt
from __future__ import division # this line is important to avoid unexpected

# This code imports your work from hw_1.py
from hw_1 import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (5, 4) # set default size of plots

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipy
%load_ext autoreload
%autoreload 2
```

```
In [ ]: # download the IMDB large movie review corpus from https://people.cs.umass.edu/~trevin/

PATH_TO_DATA = 'large_movie_review_dataset' # set this variable to point to the data
POS_LABEL = 'pos'
NEG_LABEL = 'neg'
TRAIN_DIR = os.path.join(PATH_TO_DATA, "train")
TEST_DIR = os.path.join(PATH_TO_DATA, "test")

for label in [POS_LABEL, NEG_LABEL]:
    if len(os.listdir(TRAIN_DIR + "/" + label)) == 12500:
        print ("Great! You have 12500 {} reviews in {}".format(label, TRAIN_DIR))
    else:
        print ("Oh no! Something is wrong. Check your code which loads the data")
```

```
In [ ]: # Actually reading the data you are working with is an important part of NLP

print (open(TRAIN_DIR + "/neg/98_1.txt").read())
```

Part One: Intro to NLP in Python: types, tokens and Zipf's law

Types and tokens

One major part of any NLP project is word tokenization. Word tokenization is the task of segmenting text into individual words, called tokens. In this assignment, we will use simple whitespace tokenization. Take a look at the `tokenize_doc` function in `hw_1.py`. **You should not modify `tokenize_doc`** but make sure you understand what it is doing.

```
In [ ]: # We have provided a tokenize_doc function in hw_1.py. Here is a short demo

d1 = "This SAMPLE doc has words tHat repeat repeat"
bow = tokenize_doc(d1)

assert bow['this'] == 1
assert bow['sample'] == 1
assert bow['doc'] == 1
assert bow['has'] == 1
assert bow['words'] == 1
assert bow['that'] == 1
assert bow['repeat'] == 2

bow2 = tokenize_doc("CMPSCI 585 is already my favorite class this semester!")
for b in bow2:
    print (b)
```

Now we are going to look at the word types and word tokens in the corpus. Use the `word_counts` dictionary variable to store the count of each word in the corpus. Use the `tokenize_doc` function to break documents into tokens. **You should not modify `tokenize_doc`** but make sure you understand what it is doing.

Question 1.1 (5 points)

Complete the cell below to fill out the `word_counts` dictionary variable. `word_counts` keeps track of how many times a word type appears across the corpus. For instance, `word_counts["movie"]` should store the number 61492 -- the count of how many times the word `movie` appears in the corpus.

```
In [ ]: import glob
import codecs
word_counts = Counter() # Counters are often useful for NLP in python

for label in [POS_LABEL, NEG_LABEL]:
    for directory in [TRAIN_DIR, TEST_DIR]:
        for fn in glob.glob(directory + "/" + label + "/*.txt"):
            doc = codecs.open(fn, 'r', 'utf8') # Open the file with UTF-8 encoding
            ## TODO: complete me!
```

```
In [ ]: # you should see 61492 instances of the word type "movie" in the corpus.
if word_counts["movie"] == 61492:
    print ("yay! there are {} total instances of the word type movie in the corpus")
else:
    print ("hmm. Something seems off. Double check your code")
```

Question 1.2 (5 points)

Take a look at the following values:

```
In [ ]: print ("there are {} word types in the corpus".format(n_word_types(word_counts)))
print ("there are {} word tokens in the corpus".format(n_word_tokens(word_counts)))
```

Processing math: 100%

You should see a much higher number of tokens than types. Why is that?

Answer in one or two lines here.

Question 1.3 (5 points)

Using the `word_counts` dictionary you just created, make a new dictionary called `sorted_dict` where the words are sorted according to their counts, in descending order:

```
In [ ]: # Implement me!
```

Now print the first 30 values from `sorted_dict`.

```
In [ ]: # Implement me!
```

Zipf's Law

Question 1.4 (5 points)

In this section, you will verify a key statistical properties of text: Zipf's Law (https://en.wikipedia.org/wiki/Zipf%27s_law).

Zipf's Law describes the relations between the frequency rank of words and frequency value of words. For a word w , its frequency is inversely proportional to its rank:

$$count_w = K \frac{1}{rank_w}$$

or in other words

$$\log(count_w) = K - \log(rank_w)$$

for some constant K , specific to the corpus and how words are being defined.

Therefore, if Zipf's Law holds, after sorting the words descending on frequency, word frequency decreases in an approximately linear fashion under a log-log scale.

Please make such a log-log plot by plotting the rank versus frequency **Hint: Make use of the sorted dictionary you just created..** Use a scatter plot where the x-axis is the $\log(rank)$, and y-axis is $\log(frequency)$. You should get this information from `word_counts`; for example, you can take the individual word counts and sort them. dict methods `.items()` and/or `.values()` may be useful. (Note that it doesn't really matter whether ranks start at 1 or 0 in terms of how the plot comes out.) You can check your results by comparing your plots to ones on Wikipedia; they should look qualitatively similar.

Please remember to label the meaning of the x-axis and y-axis.

Processing math: 100%

```
In [ ]: import math
import operator
x = []
y = []
X_LABEL = "log(rank)"
Y_LABEL = "log(frequency)"

# implement me! you should fill the x and y arrays. Add your code here

plt.scatter(x, y)
plt.xlabel(X_LABEL)
plt.ylabel(Y_LABEL)
```

Question 1.5 (5 points)

You should see some discontinuities on the left and right sides of this figure. Why are we seeing them on the left? Why are we seeing them on the right? On the right, what are those "ledges"?

Answer in one or two lines here.

Part Two: Naive Bayes

This section of the homework will walk you through coding a Naive Bayes classifier that can distinguish between positive and negative reviews (at some level of accuracy).

Question 2.1 (10 pts)

To start, implement the `update_model` function in `hw1.py`. Make sure to read the function comments so you know what to update. Also review the `NaiveBayes` class variables in the `def __init__` method of the `NaiveBayes` class to get a sense of which statistics are important to keep track of. Once you have implemented `update_model`, run the `train_model` function using the code below. What is the size of the vocabulary used in the training documents? You'll need to provide the path to the dataset you downloaded to run the code.

```
In [ ]: nb = NaiveBayes(PATH_TO_DATA, tokenizer=tokenize_doc)
nb.train_model()

if len(nb.vocab) == 251637:
    print ("Great! The vocabulary size is {}".format(251637))
else:
    print ("Oh no! Something seems off. Double check your code before continuing")
```

Exploratory analysis

Let's begin to explore the count statistics stored by the `update_model` function. Implement the `provided_top_n` function to find the top 10 most common words in the positive class and top 10 most common words in the negative class.

Processing math: 100%

```
In [ ]: print ("TOP 10 WORDS FOR CLASS " + POS_LABEL + ":")
        for tok, count in nb.top_n(POS_LABEL, 10):
            print ('', tok, count)
        print ()

        print ("TOP 10 WORDS FOR CLASS " + NEG_LABEL + ":")
        for tok, count in nb.top_n(NEG_LABEL, 10):
            print ('', tok, count)
        print ()
```

Question 2.2 (5 points)

What is the first thing that you notice when you look at the top 10 words for the 2 classes? Are these words helpful for discriminating between the two classes? Do you think this trend carries forward to other texts from the English language? What about other languages?

Answer in one or two lines here.

Question 2.3 (10 pts)

The Naive Bayes model assumes that all features are conditionally independent given the class label. For our purposes, this means that the probability of seeing a particular word in a document with class label y is independent of the rest of the words in that document. Implement the `p_word_given_label` function. This function calculates $P(w|y)$ (i.e., the probability of seeing word w in a document given the label of that document is y).

Use your `p_word_given_label` function to compute the probability of seeing the word “amazing” given each sentiment label. Repeat the computation for the word “dull.”

```
In [ ]: print ("P('amazing'|pos):", nb.p_word_given_label("amazing", POS_LABEL))
        print ("P('amazing'|neg):", nb.p_word_given_label("amazing", NEG_LABEL))
        print ("P('dull'|pos):", nb.p_word_given_label("dull", POS_LABEL))
        print ("P('dull'|neg):", nb.p_word_given_label("dull", NEG_LABEL))
```

Which word has a higher probability, given the positive class? Which word has a higher probability, given the negative class? Is this behavior expected?

Answer in one or two lines here.

What is the purpose of the independence assumption for the Naive Bayes classifier?

Answer in one or two lines here.

Question 2.4 (5 pts)

In the next cell, compute the probability of the word "stop-sign." in the positive training data and negative training data.

Processing math: 100%

```
In [ ]: print ("P('stop-sign.'|pos):", nb.p_word_given_label("stop-sign.", POS_LABEL))
        print ("P('stop-sign.'|neg):", nb.p_word_given_label("stop-sign.", NEG_LABEL))
```

What is unusual about $P(\text{'stop-sign.'}|\text{pos})$? Why is this a problem?

Answer in one or two lines here.

Question 2.5 (5 pts)

We can address the issues from question 2.4 with add- α smoothing (like add-1 smoothing except instead of adding 1 we add α). Implement `p_word_given_label_and_alpha` and then run the next cell. Hint: look at the slides from the lecture and the corresponding exercise on add-1 smoothing.

```
In [ ]: print ("P('stop-sign.'|pos):", nb.p_word_given_label_and_alpha("stop-sign.", POS_LABEL, alpha))
```

Question 2.6 (5 pts)

Prior and Likelihood

As noted before, the Naive Bayes model assumes that all words in a document are independent of one another given the document's label. Because of this we can write the likelihood of a document as:

$$P(w_{d1}, \dots, w_{dn} | y_d) = \prod_{i=1}^n P(w_{di} | y_d)$$

However, if a document has a lot of words, the likelihood will become extremely small and we'll encounter numerical underflow. Underflow is a common problem when dealing with probabilistic models; if you are unfamiliar with it, you can get a brief overview on [Wikipedia](https://en.wikipedia.org/wiki/Arithmetic_underflow) (https://en.wikipedia.org/wiki/Arithmetic_underflow). To deal with underflow, a common transformation is to work in log-space.

$$\log[P(w_{d1}, \dots, w_{dn} | y_d)] = \sum_{i=1}^n \log[P(w_{di} | y_d)]$$

Implement the `log_likelihood` function (Hint: it should make calls to the `p_word_given_label_and_alpha` function). Implement the `log_prior` function. This function takes a class label and returns the log of the fraction of the training documents that are of that label.

Question 2.7 (5 pts)

Naive Bayes is a model that tells us how to compute the posterior probability of a document being of some label (i.e., $P(y_d | \mathbf{w}_d)$). Specifically, we do so using bayes rule:

$$P(y_d | \mathbf{w}_d) = \frac{P(y_d)P(\mathbf{w}_d | y_d)}{P(\mathbf{w}_d)}$$

In the previous section you implemented functions to compute both the log prior ($\log[P(y_d)]$) and the log likelihood ($\log[P(\mathbf{w}_d | y_d)]$). Now, all you're missing is the *normalizer*, $P(\mathbf{w}_d)$.

Processing math: $\log[P(\mathbf{w}_d)]$

Derive the normalizer by expanding $P(\mathbf{w}_d)$.

Write your answer here using mathjaxx (similar to latex). If you are not comfortable using mathjaxx, a scanned version of your written answer is also fine.

Question 2.8 (5 pts)

One way to classify a document is to compute the unnormalized log posterior for both labels and take the argmax (i.e., the label that yields the higher unnormalized log posterior). The unnormalized log posterior is the sum of the log prior and the log likelihood of the document. Why don't we need to compute the log normalizer here?

Answer in one or two lines here.

Question 2.9 (5 pts)

As we saw earlier, the top 10 words from each class do not give us much to go on when classifying a document. A much more powerful metric is the likelihood ratio, which is defined as

$$LR(w) = \frac{P(w|y=\text{pos})}{P(w|y=\text{neg})}$$

A word with LR 3 is 3 times more likely to appear in the positive class than in the negative. A word with LR 0.3 is one-third as likely to appear in the positive class as opposed to the negative class.

```
In [ ]: # Implement the nb.likelihood_ratio function and use it to investigate the
print ("LIKELIHOOD RATIO OF 'amazing':", nb.likelihood_ratio('amazing', 0.2))
print ("LIKELIHOOD RATIO OF 'dull':", nb.likelihood_ratio('dull', 0.2))
print ("LIKELIHOOD RATIO OF 'and':", nb.likelihood_ratio('and', 0.2))
print ("LIKELIHOOD RATIO OF 'to':", nb.likelihood_ratio('to', 0.2))
```

What is the minimum and maximum possible values the likelihood ratio can take?

Answer in one or two lines here.

Find the word in the vocabulary with the highest likelihood ratio below.

```
In [ ]: # Implement me!
# Print the word with the highest likelihood ratio here
```

Question 2.10 (5 pts)

The unnormalized log posterior is the sum of the log prior and the log likelihood of the document. Implement the `unnormalized_log_posterior` function and the `classify` function. The `classify` function should use the unnormalized log posteriors but should not compute the normalizer. Once you implement the `classify` function, we'd like to evaluate its accuracy.

Processing math: 100%


```
In [ ]: print (nb.evaluate_classifier_accuracy(0.2))
```

Question 2.11 (5 pts)

Try evaluating your model again with a smoothing parameter of 1000.

```
In [ ]: print (nb.evaluate_classifier_accuracy(1000.0))
```

Does the accuracy go up or down when the pseudo count parameter is raised to 1000? Why do you think this is?

Answer in one or two lines here.

Question 2.12 (5 pts)

Find a review that your classifier got wrong.

```
In [ ]: # In this cell, print out a review your classifier got wrong, along with its
```

What are two reasons your system might have misclassified this example? What improvements could you make that may help your system classify this example correctly?

Answer here.

Question 2.13 (5 pts)

Often times we care about multi-class classification rather than binary classification.

How many counts would we need to keep track of if the model were modified to support 5-class classification?

Answer in one or two lines here.

What would be the new decision rule (i.e., how would the classify function change)?

Answer in one or two lines here.