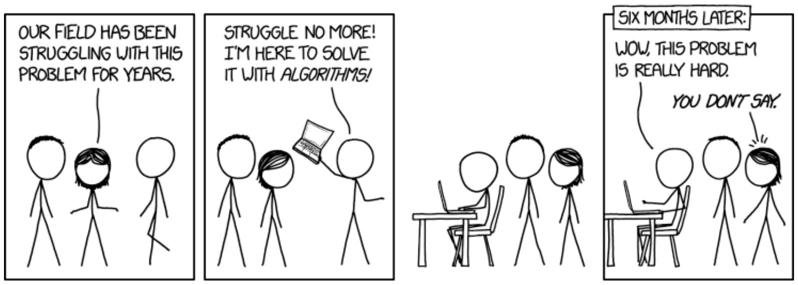
CS51A - Assignment 7

Due: Thursday March 14, at 11:59pm


OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.
STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH *ALGORITHMS!*
SIX MONTHS LATER:
WOW, THIS PROBLEM IS REALLY HARD.
YOU DON'T SAY.

https://xkcd.com/1831/

As always, read through this entire handout before starting to make sure you understand what's expected of you. Put your answers to the problems below in a file named with your first name and last name followed by `assign7.py`.

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, talk to Prof Papoutsaki or Dr. Dave and we can pair people up.

High-level Overview

For this assignment we will build a Naive Bayes text sentiment classifier that is based on online reviews and predicts whether a given review is positive or negative. As we've done with previous assignments, we're going to guide you through implementing the Naive Bayes classifier by asking you to write a collection of functions.

Training: Our trained model will consist of two dictionaries, one representing positive examples and one representing negative examples. The entries in the dictionaries will have the key be a word and the corresponding value to be the $p(word|label)$. To calculate these all you'll need to do is iterate through each file and count how many times a word occurs and then divide it by the total number of training examples (that is the total number of lines in the file), e.g.,:

$$p(word|positive) = \frac{\text{how many } \textit{positive} \text{ examples contained } \textit{word}}{\text{the total number of } \textit{positive} \text{ examples}}$$

We've setup the data so that this really comes down to calculating:

$$p(word|positive) = \frac{\text{how many times } word \text{ occurred in the } positive \text{ file}}{\text{the number of lines in the } positive \text{ file}}$$

To calculate these for each word, you'll first read through the file and store all of the individual word counts (i.e., the numerator) and then divide all of these values by the number of examples (i.e., number of lines in the file). You'll do this once for the positive examples and once for the negative (though using the same code!).

Testing: Once we have our two dictionaries of probabilities, we'll be ready to classify new examples. Given a new review to classify, $w_1, w_2, ..., w_m$, we'll calculate the probability of that review based on the model as:

$$p(label|w_1, w_2, ..., w_m) \approx p(w_1|label) * p(w_2|label) * ... * p(w_m|label) = \prod_{i=1}^{m} p(w_i|label)$$

This means that we'll multiply each of the word probabilities given the label for each word in the review. We'll do this for both classes (positive and negative) and then classify the review as the label with the highest probability.

Data

For this assignment, I've put together a collection of reviews from `www.rateitall.com`. The reviews come from a variety of domains including movies, music, books, and politicians. The orginal data have ratings from 1 to 5, but I have cleaned these up, leaving only 1s and 5s: those with a 5 score are "positive" and those with a 1 are "negative".

To get access to the review data, do the following (which should feel similar to previous assignments):

- Create a directory called `assignment7` somewhere on your computer.
- Download the following file *into your assignment7* directory and unzip it by double-clicking on it:

  `http://www.cs.pomona.edu/~dkauchak/classes/cs51a/assignments/assign7-starter.zip`
- Delete the zip file. You should now have a folder called `assignment7` with six files in it.

The dataset contains three pairs of files. For each pair, the `.positive` file contains all of the positive reviews and the `.negative` all of the negative reviews. As with previous assignments, to help in testing and debugging, I've provided `simple.positive` and `simple.negative`, which only contain three examples per file and will be useful for illustrating how different functions work. `train.positive` and `train.negative` contain the text examples that you should use to train your model. If you open up `train.positive` you'll see the first few examples are:

```
perhaps the perfect action thriller movie . funny , suspenseful dramatic packed
has it all

a well constructed thriller .

my favorite action movie yet !

i love this movie ! bought it on dvd when came out . it’s better than the
original , and jamie lee lindsey were great
```

To make life simpler for you, I’ve already preprocessed the data so that all you need to do is count word occurrences. Specifically:

- I’ve already tokenized and lowercased all of the words. I left the punctuation characters in just in case they’re indicative of a particular class (e.g., more use of exclamation points).
- I removed duplicate words from the reviews, so each review represents a set of words. You can see this in the fourth example, where it originally read “I bought it on dvd...”

To evaluate our model, we’ll use `test.positive` and `test.negative`.

Maybe not so Naive Bayes

Implement the following functions that will build up our Naive Bayes classifier functionality.

Training

1. [**3 points**] Write a function called `get_file_counts` that takes as input a filename and returns a dictionary with the number of times each word occurred in that file. Each line in the file will contain an example. I’ve already done all of the preprocessing for you, so just use the `split` function to split a line up into its individual words. For example:

   ```
   >>> get_file_counts("simple.positive")
   {'i': 3, 'loved': 3, 'it': 2, 'that': 2, 'movie': 1, 'hated': 1}
   ```

2. [**2 points**] Write a function called `counts_to_probs` that takes a dictionary and a number and generates a new dictionary with the same keys where each value has been divided by the input number. For example:

   ```
   >>> counts = get_file_counts("simple.positive")
   >>> counts_to_probs(counts, 3)
   ```

```
{'i': 1.0, 'loved': 1.0, 'it': 0.66, 'that': 0.66, 'movie': 0.33, '
hated': 0.33}
```

(I truncated the decimals so they'd fit better.)

Note that if you call this function with the word counts from a file and the number of lines int the file you'll get back the probabilities of each word (which is what we need!).

3. [**1 point**] Write a function called `train_model` that takes as input a filename containing examples and returns a dictionary with the word probabilities ($p(word|label)$). This should be a very short function that mostly just uses the previous two functions.

```
>>> train_model("simple.positive")
{'i': 1.0, 'loved': 1.0, 'it': 0.66, 'that': 0.66, 'movie': 0.33, '
hated': 0.33}
```

   Hint: we wrote a function in class for counting the number of lines in a file, which you may use if that's helpful.

Classifying

4. [**3 points**] Write a function called `get_probability` that takes as input two parameters, a dictionary of word probabilities and a string (representing a review), and returns the probability of that review by multiplying the probabilities of each of the words in the review. A few notes:

   - To determine words, just split on whitespace (i.e., use `split`).
   - Make sure to lowercase the words in the review before looking them up since our model only has lowercase words.
   - One of the key challenges with any probabilistic model is how to handle words that you've never seen before. Like in class, we're going to cheat a little bit. If you come across a word in the review that you have never seen before (i.e., don't have a probability in your dictionary) then just assume its probability is a small constant. In our case, we'll use ($1/11000 = 0.00009$).

   For example,

```
>>> pos_model = train_model("simple.positive")
>>> get_probability(pos_model, "I hated that class")
2.0202020202020202e-05
```

   The answer you get is:

```
p(i | pos) * p(hate|pos) * p(that | pos) * p(class|pos) =
    1.0    *     0.333   *      0.666    *     0.00009  = 0.00002
```

The first three words are found and the fourth is not so it is assigned the constant 1/11000 probability.

5. [**2 points**] Write a function called `classify` that takes three inputs: a string representing a review, the positive model (i.e., a dictionary of word probabilities), and the negative model (i.e., another dictionary of word probabilities). The function should return "positive" or "negative" depending on which model has the highest probability for the review. Ties should go to positive.

   Again, most of the work should be done by previous functions.

6. [**3 points**] To make it easy to play with our model, write an interactive function called `sentiment_analyzer` that takes two files as input, a positive examples file and a negative examples file, in that order. The function should train a positive and negative model using these files and then repeatedly ask the user to enter a sentence and then output the classification of that sentence (as positive or negative). A blank line/sentence should terminate the function.

   For example, here's a short transcript:

```
>>> sentiment_analyzer("train.positive", "train.negative")
Blank line terminates.
Enter a sentence: I like pizza
positive
Enter a sentence: I hate pizza
negative
Enter a sentence: I slipped on a banana
positive
Enter a sentence: I slipped on a bad banana
negative
Enter a sentence: computer science
positive
Enter a sentence:
>>>
```

Evaluation

Now that we have a working model, we should figure out how good it is. First, we can use a quantitative measure. To do this, we're going to classify our test examples and calculate what proportion we get right (called the *accuracy*).

7. [**4 points**] Write a function called `get_accuracy` that takes *four* files as input in this order:

- The positive test file (e.g., `test.positive`)
- The negative test file (e.g., `test.negative`)
- The positive training file (e.g., `train.positive`)
- The negative training file (e.g., `train.negative`)

The function should train the model (i.e., both positive and negative counts) and then classify all of the test examples (both positive and negative) and keep track of the accuracy of the model. The function should print out three scores: the accuracy on the positive test examples, the accuracy on the negative test examples, and the accuracy on all of the test examples. For example:

```
>>> get_accuracy("test.positive", "test.negative", "train.positive", "train.negative")
Positive accuracy: 0.#####
Negative accuracy: 0.#####
Total accuracy: 0.#####
```

(I've hidden the actual values printed out since I want you to be surprised when you get your code running :)

Advice:

- You can either write this as a single function or write helper function(s) to help you do some of the work.
- This will be one of the longer functions we've written so make sure to think about what you want to do and try and test as you go.
- Remember you can use `print` statements to help you understand what your code is doing, but remove these before submitting your work.
- Try testing on the simple examples for debugging with a call like:

```
>>> get_accuracy("simple.positive", "simple.negative",
"simple.positive", "simple.negative")
Positive accuracy: 1.0
Negative accuracy: 0.6666666666666666
Total accuracy: 0.8333333333333334
```

(Testing on your training test like this isn't a proper way of evaluating a model, but it's good for debugging purposes.)

8. [**4 points**] Include 1-2 short paragraphs (less than half a page, though), as either comments or a triple quoted string, at the end of your file evaluating the quality of your Naive Bayes model. Your discussion must include:

- The output of `get_accuracy` on the test examples and a discussion of these results.

  - At least one positive and one negative example that you make up where the model makes the wrong decision. Play with the interactive version to find these.
  - A concrete discussion of why the model makes a mistake. This should include you looking up the word probabilities in the model and trying to understand what word(s) are causing it to make a mistake.

When you're done

Make sure that your program is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- You should have comments delimiting the two sections.
- Each function should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

In addition, make sure that you've used good *style*. This includes:

- Following naming conventions, e.g., all variables and functions should be lowercase.
- Using good variable names.
- Good use of booleans. You should NOT have anything like:)

  ```
  if boolean_expression == True:
  ```

  or

  ```
  if boolean_expression == False:
  ```

  instead use:

  ```
  if boolean_expression:
  ```

  or

  ```
  if not (boolean_expression): # or some other way of negating the expression
  ```

- Proper use of whitespace, including indenting and use of blank lines to separate chunks of code that belong together.
- Make sure that none of the lines are too long, i.e., cross the red line in Wing.

Submitting

You only need to submit your `.py` file (and not any of the starter that you downloaded). Submit your `assign7.py` file online using the courses submission mechanism under “assign7”.

Grading

	points
Training	6
Classifying	8
Evaluation	8
Comments, style	3
total	25