

CIS 530 Fall 2009 HW 1

Instructor: Mitch Marcus

TA: Constantine Lignos

Released: September 22nd, 2009

Revised: September 27th, 2009

Due: October 8th, 2009 at 4PM

Overview

This assignment will give you practice in the basics of using Python and NLTK and introduce you to conditional probability in some simple NLP tasks. While this is a programming assignment, the focus is on NLP, not programming itself. Your solutions should be short and simple, but do not sacrifice readability to reduce your line count.

The length of an example solution is given with each problem to give you a point of reference. You should not worry about matching this length, but if your solution is much longer you are probably overcomplicating things. Solution lengths include any helper functions used and include all comments and line breaks needed to make the code readable.

For all problems, you need not do any more preprocessing/filtering of data than is explicitly requested in the problem. For example, when you are asked to examine the characters of a language, you should look at all characters in all tokens, even non-alphabetic ones. Note that many of the problems will request that you lowercase tokens/characters before counting them; this is very common practice in NLP.

Submitting your work

Code Organization

Please submit all your code for the assignment in a single file called “hw1_yourpennkey.py” where “yourpennkey” is your PennKey.

Submitting files

To electronically submit homework, if you’re not already working on Eniac you need to place the file containing your solution on your SEAS account storage. One way to do this is using an SFTP client, for example, scp:

```
% scp <filename> eniac.seas.upenn.edu:  
Password: *****
```

Then connect via ssh to seas.upenn.edu and use the turnin command to submit your file for grading:

```
% turnin -c cis530 -p hw1 <filename>
```

This should print out a confirmation message. You can run turnin multiple times before the deadline, but each time you run turnin, it overwrites your previous submission for that assignment. You can check that the homework submitted successfully:

```
% turnin -c cis530 -v
```

This will show a list of the file(s) you have submitted.

1 Counting is Key

1.1 Roll Your Own (15 Points)

One of the most basic tasks in NLP code is counting the occurrences of input samples. The built-in dictionary type in Python is a very useful way to do this. Write a function `count_words(words)` that takes a sequence of words and returns a dictionary where each word in the input is a key and the corresponding value is the number of times that word appeared in `words`.

Here's how you would use this function:

```
>>> words = ['and', 'when', 'she', 'shines', 'she', 'really', 'shows', 'you', 'all',
'she', 'can', 'oh', 'rio', 'rio', 'dance', 'across', 'the', 'rio', 'grande']
>>> counts = count_words(words)
>>> counts['rio']
3
>>> counts['and']
1
```

The documentation for the dictionary type is surprisingly difficult to locate, you can find it at <http://docs.python.org/library/stdtypes.html#mapping-types-dict>.

You may have noticed that NLTK has a class for accomplishing this task, called a `FreqDist` (short for frequency distribution). As the goal of this problem is to give you practice with Python's built-in types, you may not use that class (or any other class NLTK provides) in your solution for this question, but you may want to compare your function's output with that of a `FreqDist` to test that your code works.

Example solution length: 11 lines

1.2 Use NLTK (5 Points)

Now that you've done it the hard way, use NLTK to make your life easier (and continue to do so for the rest of the assignment). Write a function `count_words_easy(words)` that takes the same input as `count_words` and returns a `FreqDist` corresponding to the input words. Although you're returning a `FreqDist` and not a dictionary, you'll notice that you can use the result just as above to get the number of occurrences for a word.

Example solution length: 3 lines

2 Counting Suffixes (20 Points)

An important problem in computational linguistics is morphological analysis. This consists of breaking down a word into its component pieces, for example "losses" might be broken down as "loss + es". In English, morphology is relatively simple and is mostly comprised of prefixes and suffixes. To get an idea of what suffixes are common in English (and thus could be morphemes), we can look at the frequencies of the last two characters of sufficiently long words.

Write a function `top_suffixes(words)` that takes a sequence of words as an input and returns the 10 most frequent two-character suffixes of the input words. We define a two-character suffix as the last two characters of any word of length 5 or more, thus your function may simply ignore any word shorter than five characters. Your function should use the NLTK `FreqDist` class to count these suffixes. The function would be used like this:

```
>>> words = ['caraa', 'bataa', 'dadbb', ...]
>>> top_suffixes(words)
['aa', 'bb', ...]
```

As a sanity test, you may want to test your function on the words in Jane Austen's book *Emma*. You can load the words of *Emma* as follows:

```
emma_words = nltk.corpus.gutenberg.words('austen-emma.txt')
```

If you pass the words of *Emma* (or any sufficiently large English document) as the argument to your function, you should see some common English suffixes in the output.

Example solution length: 6 lines

3 Identifying a Language

3.1 The Top Characters (15 Points)

Let's say you're given a document and you want to know what language it's written in. One way to guess would be to collect some very simple statistics about the text of the document. In this case, we're going to learn the conditional probabilities of individual characters given a language and then see how they compare.

Write a function `top_chars(langs)` that takes a sequence of names of languages and returns the top 5 most probable characters in each language, determined by analyzing each language's version of The Universal Declaration of Human Rights (UDHR). To make sure we can easily compare the text of different languages, we need to make sure they're all in the same Unicode text encoding. A side effect of this is that if you print the output of your function, 'u' character will precede every string to let you know that the encoding is Unicode. For example, the usage would look like this if the most probable characters were 'abcde' in English and 'fghij' in German:

```
>>> langs = ['English', 'German_Deutsch']
>>> top_chars(langs)
[[u'a', u'b', u'c', u'd', u'e'], [u'f', u'g', u'h', u'i', u'j']]
```

Again, the answer returned above is properly formatted but not the correct ranking for either language, so your results should be different.

To get the words from the UDHR in a particular language in the desired encoding, consider this example:

```
>>> lang = 'English'
>>> nltk.corpus.udhr.words(lang + '-Latin1')
[u'Universal', u'Declaration', u'of', u'Human', ...]
```

The '-Latin1' suffix to the language tells NLTK the encoding you want the text to be returned in, do not change this.

To calculate the probabilities for each character, use NLTK's `ConditionalFreqDist` class and count the lowercased version of each character in each word, conditioned on the language. You will create a single conditional frequency distribution with one condition for each language. Given the next part of this problem, you will probably want to create a helper function that returns a conditional frequency distribution given a set of languages, and then call that from a function that returns the most probable characters in each language given the conditional frequency distribution.

Example solution length: 13 lines

3.2 Mind the Gap (15 Points)

As we learned in class, Kullback-Leibler divergence is a way to measure the divergence between two probability distributions. Perhaps a more scientific way to compare the character probability distributions we've generated would be to calculate the K-L divergence between them.

Write a function `compare_langs(langs)` that calculates the K-L divergence of the character probability distributions of the two languages passed in as `langs`¹. Treat the first language in `langs` as the source of the P distribution and the second language as the source of Q , and compute logarithms using base 2.

Unfortunately, since not all characters appear in all languages, and furthermore not all characters in a language appear in every document containing that language, by the strictest definition the K-L divergence of two character distributions is often infinite only because one character has zero probability in one language/document and non-zero probability in another. While this would be a great opportunity for smoothing (see <http://www.cs.bgu.ac.il/~elhadad/nlp09/KL.html> if you're curious), we're going to introduce an ugly simplification to get you through this problem: **only include a term in the sum for K-L divergence if $P(i)$ and $Q(i)$ are non-zero.**

So in our case, this means we will iterate over the possible characters in the first language (corresponding to P) and only count a character in the sum if it has been observed in the first and second languages. This means our hacked version of K-L divergence is not correct, but will still be useful. Hacks like these are a recurring theme in NLP.

Here's how the function could be used, although the value returned here is not the real answer:

```
>>> langs = ['English', 'German_Deutsch']
>>> compare_langs(langs)
.12345678
```

Hint: One good way to test your function is to compare the K-L divergence of English against two different languages. Which do you think will produce the greater divergence, English and German, or English and Swaheli? What should the K-L divergence of English and English be?

Example solution length: 18 lines (not counting helper function created in previous problem)

3.3 Extra Credit- Cracking the Code (15 Points)

Assuming you know what language a message is in, the Caesar Cipher (http://en.wikipedia.org/wiki/Caesar_cipher) can be easily cracked by aligning the most common character in the encrypted text with the most common character in the source language. For example, in English the most common character is generally considered to be 'e', so if in our encrypted text the most common character is 'i,' the cipher key is probably 4, and thus we can shift all characters by 4 positions to decrypt the message.²

The Caesar Cipher is a very simple example of a substitution cipher. Assume a more complicated cipher that's specified by an arbitrary one-to-one mapping of characters. To encrypt/decrypt it, you need to know what the substitution is for each character, i.e. "replace 'a' with 'e,' 'r' with 'q'..." for all characters of the alphabet. If you had a large enough set of encrypted and unencrypted text from the same language, you can crack the encryption cipher by simply aligning the most frequent characters in each set of words.

Write a function `align_chars(plain_words, encrypted_words)` that takes two sequences of words and cracks the cipher by aligning the character distributions of the lowercased characters in those words. It returns a sequence of tuples of the form `(unencrypted char, encrypted char)`, with the most frequent characters first. So if the most frequent three characters (in descending order) in `plain_words` are 'a,' 'b,' 'c,' and the most frequent in `encrypted_char` are 'q,' 'e,' 'd,' the output would be:

¹Note that `compare_langs(langs)` requires `langs` be a sequence of two strings, while for `top_chars(langs)` it can be any number of strings.

²Unfortunately, code cracking is never quite that simple. We assume that the most common character in English and in the unencrypted message are the same. If the message is short or in some way unrepresentative of English in general, we could be wrong in our assumption.

```
>>> plain_words = ['hello', 'world', ...]
>>> encrypted_words = ['jfiiod', 'kajl', ...]
>>> align_chars(plain_words, encrypted_words)
[('a', 'q'), ('b', 'e'), ('c', 'd'), ...]
```

Example solution length: 12 lines

4 Identifying a Topic

4.1 The Curse of Common Words (15 Points)

Another handy use for conditional probabilities is in information retrieval, where we might be interested in what words tend to be most predictive of a particular type of document category. For example, we might predict that the word “mammogram” would have a high probability given that the document is about cancer.

Write a function `basic_conditional_words(categories)` that computes the conditional probabilities of words appearing in each category in the sequence `categories` and returns a sequence where each element is a sequence containing the 20 most probable words for each category. In other words, your return value will have as many elements as `categories`, and each of those elements will be a sequence of the 20 top words in a particular category. For example:

```
>>> categories = ['barley', 'coffee']
>>> basic_conditional_words(categories)
[['barley', 'top', 'words', ...], ['coffee', 'top', 'words', ...]]
```

To compute these statistics, load the words of a particular category as shown below and create a conditional distribution that conditions the probability of a word in the corpus given that category. You should lowercase all words from the corpus before counting them. We’ll use the Reuters corpus to get words from articles about each category like so:

```
>>> category = 'barley'
>>> nltk.corpus.reuters.words(categories=category)
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
```

Example solution length: 22 lines

4.2 A Better Way (20 Points)

You’ll notice that the output of this isn’t very useful as-is. Think about why, but you need not write it down. It would seem a better thing to do than just look at the conditional probability of a word given a category would be to create a ratio of the conditional probability of the word given the category to the base probability of the word itself.

Write another function, `better_conditional_words(categories)` that treats its argument the same but instead of simply returning the 20 most probable words for each category it returns the 20 words in each document with the highest ratio of probability given the category to overall probability. That is, it returns the top 20 words in each document where each word w is ranked by this value:

$$\frac{p(w|category)}{p(w)}$$

We calculate $p(w|category)$ as the frequency of the word among the words for that category, and we calculate $p(w)$ over the entire Reuters corpus, which can be obtained using `nltk.corpus.reuters.words()` with no arguments.

Example solution length: 16 lines (not counting helper function created in previous problem)

5 Debriefing

Include the answers to these questions in comments at the end of your file.

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. How deeply do you feel you understand the material it covers (0%-100%)?
4. Any other comments?

These questions are intended to help us calibrate the homework assignments. Your answers will not affect your grade, and unless you request it we will not contact you about them.

Revision History

9/22/2009	Initial version
9/24/2009	Added example solution lengths and a usage example for <code>compare_langs(langs)</code> .
9/25/2009	Fixed <code>basic_conditional_words(categories)</code> definition to return a value instead of printing a result and made the format of its return value more clear.
9/27/2009	Changed all references to 'letters' to 'characters' and added a note in the overview about not filtering tokens/characters unless explicitly requested.