Homework 6

1. Smoothed language model

a. Define a class called `SmoothedLangModel`. It should implement the smoothing method described in Handout 7, #12. The constructor should take two arguments: a text[1] and a value `epsilon`.

Define a method `uprob` that takes a word x as input and returns its smoothed probability $\tilde{p}(x)$. Also define a method `biprob` that takes two words x, y as input and returns the smoothed probability $\tilde{p}(y|x)$.

Tip: create a `FreqDist` and `ConditionalFreqDist` in the usual way,[2] and compute the probabilities as needed, using the formulae of HO 7, #11c. Do not try to precompute smoothed probabilities: the matrix is too big. However, it would be reasonable to precompute the constants $n\epsilon$ and $N + n^2\epsilon$.

Examples:

```
>>> lm = SmoothedLangModel('abbdabdbbd', 0.5)
>>> lm.uprob('a')
0.2413793103448276
>>> lm.biprob('a','b')
0.7142857142857143
>>> lm.pibrob('b','a')
0.07692307692307693
```

Note: if you reload the class definition, you will need to create the instance `lm` over again. The old instance will continue to use the old class definition.

b. If we access a FreqDist with a non-existent key, it does not signal an error, but returns 0. This is the wrong behavior for the smoothed language model: when we normalize, we count the epsilons for the *existing* types, not for random unanticipated types. Revise `uprob` and `biprob` to check whether the words they are given as input are valid. If not, signal an error. For example:

```
>>> lm.biprob('a', 'c')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Illegal word: c
```

Here is how to signal an error:

```
>>> raise Exception('Illegal word: {}'.format('c'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Illegal word: c
```

[1] As a general rule, I use the term "text" to mean a list or other iterable containing tokens, not necessarily an NLTK `Text` object.

[2] Be sure to use our re-definition of `bigrams`, not `nltk.bigrams`.

Caution: you should check both words against the unigram distribution, not the bigram distribution. The unigram distribution contains the complete list of types.

2. Add the following methods to `SmoothedLangModel`.

a. Methods `ucost` and `bicost` that return the negative log of `uprob` and `biprob`, respectively.

```
>>> lm.ucost('a')
0.6172999578846993
>>> lm.bicost('a','b')
0.146128035678238
>>> lm.bicost('b', 'a')
1.1139433523068367
```

b. Method `cost` that takes a text and returns its average cost per word. (N.b.: average cost, not total cost. For the sake of comparison to the handout, note that $.322 \times 4 = 1.29$ and $.681 \times 4 = 2.72$.)

```
>>> lm.cost('bbda')
0.3218864527397058
>>> lm.cost('bbad')
0.6806173464710077
```

3. Define a class called `UnigramLangModel`, which uses unigram frequencies only, ignoring context. Its constructor should take a single argument: a text. Define a method `cost` that takes a text as input and returns the average cost per word. Example:

```
>>> ulm = UnigramLangModel('abbdabdbbd')
>>> ulm.cost('bbda')
0.45597718523607966
>>> ulm.cost('bbad')
0.45597718523607966
```

4. Define a function named `split_text` that takes two arguments: a text and a proportion p between 0.0 and 1.0. The proportion p indicates how much of the text to use as training; the remainder is used for testing. The return value is a pair `(train, test)`, in which `train` and `test` are token lists. For example:

```
>>> split_text(list('abcdefgh'), .75)
(['a', 'b', 'c', 'd', 'e', 'f'], ['g', 'h'])
```

Fetch *Moby Dick* from the Gutenberg corpus, and split it into 90% training data and 10% testing data. Store the training portion in the variable `mtrain`, and store the testing portion in `mtest`.

5. We would like to train each model on `mtrain` and test it on `mtest`, but there's a problem: there will assuredly be words in the second half that

were never seen in the first half, which will cause an error when we test the language model. We will take the approach of transforming the data to avoid the problem.

We will use the training data to construct a "filter" that can be applied to any text. In brief, the filter will use the training data to determine a *vocabulary* of words that occur n or more times in the training data (for a given minimum count n), and when the filter is applied, it replaces all tokens that are not in the vocabulary with the special token `'UNK'`.

a. Define a class called `FreqFilter`. Its `__init__` method should take two arguments: a training text and a minimum count n. It should downcase all the words in the training text, but it should not throw any tokens away. It should construct a FreqDist from the tokens after downcasing, and then it should construct the set of words whose count is at least n. Finally, it should store the resulting set of words in the member `vocabulary`. For example:

```
>>> filt = FreqFilter('AaBbcd', 2)
>>> filt.vocabulary
{'b', 'a'}
```

b. Define a method called `apply` that takes a text as input, and returns a transformed text as follows. First, downcase the tokens, but do not throw any away. Then look up the tokens in the vocabulary. If the token is found, leave it as is, but if it is not in the vocabulary, replace it with `'UNK'`. For example:

```
>>> filt.apply('BaAc')
['b', 'a', 'a', 'UNK']
```

6. Comparing language models

a. Construct a `FreqFilter` from `mtrain`, with $n = 2$, and call it `mfilt`. Apply it to `mtrain` and store the result as `fmtrain`. Also apply the filter to `mtest` and store the result as `fmtest`. Then use `fmtrain` to train a `SmoothedLangModel` with $\epsilon = 0.5$. Test the model on `fmtest` and store the test average cost in `slm_cost`. Similarly train and test a `UnigramLangModel`, and store its test average cost in `ulm_cost`.

b. You will notice that the unigram model actually outperforms the smoothed bigram model. Train a new smoothed model with $\epsilon = 0.01$, and store its test average cost in `slm01_cost`.

c. **Discussion.** Obviously the value for ϵ matters. Is it suprising that the smoothed model performs worse than the unigram model at the "default" value of $\epsilon = 0.5$? What might the cause be?

7. Text classification

a. Fetch the following texts from the NLTK Gutenberg corpus: Jane Austen's *Emma, Persuasion,* and *Sense and Sensibility;* and Chesterton's *The Wisdom of Father Brown* and *The Man Who Was Thursday.* Use *Emma* as training data. Construct a `FreqFilter` with $n = 2$, and train a smoothed language model with $\epsilon = .01$. Then, for each of the remaining four texts, apply the filter and test the language model. Store the list of test average costs (four values, in the order in which the texts are listed above) in the variable `scores`.

b. **Discussion.** If we think of the test score as a measure of similarity to Austen's writing, do we successfully rank Austen's novels above Chesterton's?