Project Semester 1
Estimating a Language Model to Generate Reviews

1. *Estimating the Language Model*

The objective of this exercise is to build a program capable of automatically writing and evaluating food product reviews like the ones found in the Amazon Review corpus (here).

A language model of order n allows us to determine $p(w_i|w_{i-1}, w_{i-2}, \dots, w_{i-n+1})$, the probability of finding word w_i after observing the previous $n-1$ words, and, by extension, allows us to then determine the probability of a sentence.

The probabilities (aka parameters) of the language model will be estimated from the Prime Pantry reviews which can be found in the project folder on github.

The file test_reviews.txt is provided for you to test your program on a smaller, more manageable sample of data. It is recommended that you try to implement the program by hand using simple examples before using the full dataset of product reviews (Prime_Pantry_train.txt).

a. Let w_i be the word at the ith position of a sentence. How is the probability of observing the word w_i knowing the two previous words w_{i-1} and w_{i-2} estimated for example ?

b. Write the function *make_ngrams* which returns the list of successive *n*grams from a string of words.
For example, when the string "I love chocolate ice-cream." is passed to this function with the argument *n*=3, it should return the list :
[("I", "love", "chocolate"), ("love", "chocolate", "ice-cream"), ("chocolate", "ice-cream", ".")].

c. Preprocess each input review by adding special tokens to the beginning and the end of the review. The token <s> should be added to the start *n-1* times, and the token </s> should be appended. Why do you think this is useful ?

d. Write a function *make_conditional_probas* that estimates all of the probabilities for your model from a file passed as argument. One way of doing this is by first constructing a count table which looks like the following, when estimating a trigram model:

```
{('BEGIN', 'NOW'): defaultdict(<class 'int'>, {'I': 2}),
 ('I', 'do'): defaultdict(<class 'int'>, {'not': 1}),
 ('I', 'like'): defaultdict(<class 'int'>, {'chocolate': 1}),
 ('NOW', 'I'): defaultdict(<class 'int'>, {'like': 1, 'do': 1}),
 ('chocolate', 'ice-cream'): defaultdict(<class 'int'>, {'.': 1}),
 ('chocolate', 'pudding'): defaultdict(<class 'int'>, {'.': 1}),
 ('do', 'not'): defaultdict(<class 'int'>, {'like': 1}),
 ('ice-cream', '.'): defaultdict(<class 'int'>, {'END': 1}),
 ('like', 'chocolate'): defaultdict(<class 'int'>,
                                    {'ice-cream': 1,
                                     'pudding': 1}),
```

The keys represent (*a*,*b*), the bigrams contained within each trigram. Their values are dictionaries which have as keys *c* (the following word) and as values the frequency of *c* following *a*,*b* .

Once the different counts have been established, it is possible to sum over all of the counts to obtain a total and to then divide each individual count by this value.

$$P(c|a,b) = \frac{count(a,b,c)}{total = \sum_{i=1}^{n}(a,b,c_i) = count(a,b)}$$

The output probability table should look something like this:

```
{('BEGIN', 'NOW'): {'I': 1.0},
 ('I', 'do'): {'not': 1.0},
 ('I', 'like'): {'chocolate': 1.0},
 ('NOW', 'I'): {'do': 0.5, 'like': 0.5},
 ('chocolate', 'ice-cream'): {'.': 1.0},
 ('chocolate', 'pudding'): {'.': 1.0},
 ('do', 'not'): {'like': 1.0},
 ('ice-cream', '.'): {'END': 1.0},
 ('like', 'chocolate'): {'ice-cream': 0.5, 'pudding': 0.5},
 ('not', 'like'): {'chocolate': 1.0},
 ('pudding', '.'): {'END': 1.0}})
```

Create parameters using the Prime_Pantry_train.txt file for *1*-gram, *2*-gram, *3*-gram, *4*-gram language models. You may serialize these parameters and load them later with the `pickle` module. For *1*-grams, this function should simply return a dictionary with all unigrams as keys and their relative frequencies as values.

2. Generation

Once the language model's parameters have been estimated, new sentences can be generated as follows: the i^{th} word of the sentence is chosen according to probability $p(w_i|h)$ where h represents the history, composed of the previous *n-1* (w_{i-2}, w_{i-1} in the case of trigrams for example). The history is then updated $h \leftarrow w_{i-1}, w_i$ and the procedure is repeated until the token </s> is generated.

Given a conditional probability distribution for a particular *n-1*-gram it is possible to generate the next token a_i with its associated probability p_i using the following function (*distrib* is a dictionary $\{a_1:p_1, \ldots, a_n:p_n\}$):

```python
import numpy as np

def sample_from_discrete_distrib(distrib):
    words, probas = zip(*distrib.items())
    probas = np.asarray(probas).astype('float64')/np.sum(probas)
    return np.random.choice(words, p=probas)
```

As an example, if the dictionary {'ice-cream': 0.5, 'pudding': 0.5} represents the probability distribution, this function will generate *ice-cream* 50% of the time and *pudding* the remaining 50%.

a. How should you initialize the history ?

b. Implement this algorithm by creating a function called *generate* which takes as argument a probability table. What do you think of the sentences obtained ? Compare the outputs for the different *n*-gram models.

c. Using Prime_Pantry_test.txt, estimate the perplexity of each test review. To keep this toy project simple, the test reviews are guaranteed to contain any ngram seen in the training corpus.

3. Refine the model

The language model's generalization capability can be improved with a few tweaks.

One of them is to add an <unk> token to the vocabulary, to allow for unkown tokens in the test reviews. You can do this by replacing tokens in the training reviews which occur only once (or an arbitrarily low number of times) with the <unk> token and then computing your parameters as before. When evaluating, you can replace test review tokens which aren't in the training vocab by <unk>.

What if a parameter still cannot be found during evaluation? One simple way to remedy this is by implementing 'stupid' backoff, a simple technique usually used for large-scale ngram models. The idea is that if you cannot find a parameter estimate for a context of *n*, then back off to a context of *n-1*, then *n-2* etc. looking at the parameter estimates of a lower order ngram model each time. If you cannot estimate using the *2*-gram model, simply use the *1*-gram estimate. This can be done recursively until you reach *1*-gram estimates, with a constant reduction factor of 0.4 applied each time.

$$S(w_i \mid w_{i-k+1}^{i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-k+1}^{i})}{\text{count}(w_{i-k+1}^{i-1})} & \text{if } \text{count}(w_{i-k+1}^{i}) > 0 \\ 0.4\,S(w_i \mid w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

$$S(w_i) = \frac{\text{count}(w_i)}{N}$$