Assignment 08:
Experiments with German N-grams

handed out: February 2nd, 21:00
to be submitted by: February 9th, 21:00

In this assignment, you are going to use conditional probability distributions over word N-grams extracted from a collection of German sentences to produce random (and sometimes amusing) German sentences. As a preview, here are some random "sentences" generated from trigrams from our solution:

- "Maria schlug eine Wärmflasche in ein Verzeichnis."
- "Der Winterschlussverkauf hat wieder geheiratet."
- "Dieser Wein ist stark wie eine Münze werfen."
- "Toms Äußerung muss wieder Überstunden gesammelt und getrocknet hat."
- "Unsere Epoche ist eine enigmatische, seltsame und anspornende Dichterin."

As a reminder: an **N-gram** is a sequence of N consecutive tokens in a sequence, which we are going to model as tuples of strings. To extract the N-grams from a sentence, the option we take here is to append N-1 dummy tokens to both the beginning of the sentence (`"BOS"` = "beginning of sentence") and to the end (`"EOS"` = "end of sentence").

With this extension, the bigrams (2-grams) in the sentence "This is the second last assignment.", tokenized to `["This", "is", "the", "second", "last", "assignment", "."]`, will be `("BOS","This")`, `("This","is")`, `("is","the")`, `("the","second")`, `("second","last")`, `("last","assignment")`, `("assignment",".")`, and `(".","EOS")`.

The trigrams (3-grams) are `("BOS","BOS","This")`, `("BOS","This","is")`, `("This","is","the")`, `("is","the","second")`, `("the","second","last")`, `("second","last","assignment")`, `("last","assignment",".")`, `("assignment",".","EOS")`, and `(".","EOS","EOS")`.

The code you will write in this assignment should be general enough to support any $N > 1$.

Task 1: A Class for Storing N-gram Models [2 points]

Your first step is to define a class which will be used to model the N-gram model over a text. The class must be called `Ngram`, and it needs a constructor with arguments `filename` (the path to a file from which the model will be extracted) and `n` (representing N). The default value for `filename` must be the empty string, and `n` will be zero if not specified otherwise. In addition to the filename and N, the class must have three dictionaries as instance variables, which need to be initialized by the constructor, but are going to be filled during the later tasks. Their names must be `raw_counts`, `prob`, and `cond_prob`.

Example usage:

```
>>> ngram_model = Ngram("de-sentences-tatoeba.txt", 2)
>>> ngram_model.n, ngram_model.filename
(2, "de-sentences-tatoeba.txt")
>>> ngram_model.raw_counts, ngram_model.prob, ngram_model.cond_prob
({}, {}, {})
```

Task 2: Extracting N-gram Counts [3 points]

The first step towards modeling the conditional distribution is to extract the raw N-gram counts from the sentence collection. This time, we are providing you with an adapted smart tokenization function `tokenize_smart(sentence)` which already splits the input sentences well enough into tokens. Use this function in a method `extract_raw_counts()` of your class, which fills the dictionary assigned to the `raw_counts` instance variable with raw N-gram counts. The logical structure of your code should be as follows:

```
for each line in the file
    cut off the trailing newline character
    tokenize the sentence (= the line) using the function provided
    append N-1 instances of "BOS" and "EOS" to the beginning and the end of the token list
    starting at each position i in the token list
        put the tokens from i to i+N into a new tuple (= the N-gram)
        increase the count of the N-gram in raw_counts by 1
        (adding the N-gram as a key if previously not present)
```

Example usage:

```
>>> ngram_model.extract_raw_counts()
>>> ngram_model.raw_counts[("kaltes","Land")]
3
>>> ngram_model.raw_counts[("schönes","Land")]
11
```

Task 3: Computing N-gram Probabilities [1 points]

The next step is to convert the raw frequency counts into probabilities. For the implementation of the class method `extract_probabilities()`, you need to compute the sum of all raw counts (= the total number of N-grams in the sentence list), and then fill the dictionary assigned to the `prob` instance variable with the raw counts divided by the sum of all counts.

Example usage:

```
>>> ngram_model.extract_probabilities()
>>> ngram_model.prob[("kaltes", "Land")]
6.794386025306823e-07
>>> ngram_model.prob[("schönes", "Land")]
2.4912748759458353e-06
```

Task 4: Computing Conditional Unigram Probabilities [3 points]

Now we can build the model giving us the conditional probabilities of the next token given the N-1 previous tokens. The implementation of the required class method `extract_conditional_probabilities()` amounts to building and storing a probability distribution over all possible following tokens for each N-1-gram which occurred in the corpus. The lookup structure will be stored in the value of the `cond_prob`

instance variable. For our example with $N = 2$, looking up how likely the token "viele" is after "sehe" is done via `model.cond_prob[("sehe")]["viele"]`. (BTW: the notation `(token)` is necessary in the bigram case to enforce that a unary tuple is looked up, because `(token) == token` in Python). The recommended logical structure of your implementation is as follows:

```
for each N-gram contained as a key in prob
    split the N-gram into the first N-1 tokens (the "mgram") and the final unigram
    if the mgram is not yet a key in cond_prob, store a new dictionary under that key
    set the value for the unigram in cond_prob[mgram] to the count of the N-gram
for every dictionary in the values of cond_prob
    add up the values assigned to all unigram keys
    divide the value under each unigram by the sum of values
```

Example usage:

```
>>> ngram_model.extract_conditional_probabilities()
>>> ngram_model.cond_prob[("beobachteten")]
{',': 0.14285714285714285,
 '.': 0.14285714285714285,
 'Erwärmung': 0.07142857142857142,
 'Tom': 0.07142857142857142,
 'das': 0.07142857142857142,
 'den': 0.07142857142857142,
 'die': 0.21428571428571427,
 'mich': 0.14285714285714285,
 'sie': 0.07142857142857142}
>>> ngram_model.cond_prob[("schönes")][("Land")]
0.03806228373702422
```

Task 5: Generating Random Tokens In Context [1 points]

Finally, we have the probability distribution that can be used to sample plausible next tokens given the previous $N-1$ tokens. Implement this functionality in a new class method `generate_random_token(mgram)`, which takes a $N-1$-gram in the tuple encoding, and randomly generates a plausible next token by sampling from the probability distribution you stored in `cond_prob[mgram]`. In Python 3.6 or higher (the version you should be using), this can easily be done using the `choices()` function from the package `random`. Compare the documentation, or the slides of Session 11, for usage examples. All you really need to do for this task is to prepare two lists, and feed them to `choices()` as arguments.

Example usage:

```
>>> ngram_model.generate_random_token(("den",))
'Tag'
>>> ngram_model.generate_random_token(("den",))
'Hund'
>>> ngram_model.generate_random_token(("den",))
'Staub'
```

Task 6: Generating Random Sentences [2 points]

The last step is to implement the method `generate_random_sentence()` for generating a random sentence with the help of the `generate_random_token(mgram)` function. The key idea is to initialize the sentence with a list of $N-1$ instances of `"BOS"`, and adding random words based on the last $N-1$ of the current list until `"EOS"` is generated for the first time. Removing the `"BOS"` and `"EOS"` dummy tokens from the resulting

list gives you the final sentence to return. You can use the provided helper function `list2str(sentence)` to format the generated sentence list as a string.

Example usage:

```
>>> list2str(ngram_model.generate_random_sentence())
'Riga, daß die Nacht wach geworden sind schöne Seele ernährt für Sprachen
zu schneien.'
>>> list2str(ngram_model.generate_random_sentence())
'Ähh, und von der anderen Entscheidung ist viel Zeit vergessen.'
```

That's it! before submitting, don't forget to test your functions one last time using `test_ex_08.py`! Also, you might want to test your model for $N = 3$ as well, as this is going to be used by the additional tests.

Total points: 12