Denny Ceccon

2 Followers About Follow

A simple numerical example for Kneser-Ney Smoothing [NLP]

Denny Ceccon Feb 2, 2019 · 7 min read

As I was working my way through a Natural Language Processing project, I came to the idea of *Kneser-Ney Smoothing*. I won't get into the details here, but the thing is the main equation is pretty bad-looking. In order to make some sense of that, I searched for numerical examples, but, surprisingly, could find none. And I thought by now we could find anything online… "Well, why don't I write my own?", I figured.

(A few months later, I thought "Why not make it public, for other sufferers out there?" And here we are.)

It all starts with the Kneser-Ney Probability equation (as in here, eq. 4.35), a recursive formula that calculates the probability of a word given previous words, as based on a corpus:

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max\left(c_{KN}(w_{i-n+1}^{i}) - d, 0\right)}{c_{KN}(w_{i-n+1}^{i-1})} + \lambda\left(w_{i-n+1}^{i-1}\right) P_{KN}(w_i|w_{i-n+2}^{i-1})$$

Let's call this Scary Function.

(You might notice some small changes in mathematical notation if you compare to the source material I linked, but I am using the above form because it seems to me to be the standardized notation.)

We start this example with an example text (from here):

```
exampleText = "Paragraphs are the building blocks of papers. Many
students define paragraphs in terms of length: a paragraph is a
group of at least five sentences, a paragraph is half a page long,
etc. In reality, though, the unity and coherence of ideas among
sentences is what constitutes a paragraph. A paragraph is defined as
“a group of sentences or a single sentence that forms a unit”.
Length and appearance do not determine whether a section in a paper
is a paragraph. For instance, in some styles of writing,
particularly journalistic styles, a paragraph can be just one
sentence long. Ultimately, a paragraph is a sentence or group of
sentences that support one main idea. In this handout, we will refer
to this as the “controlling idea,” because it controls what happens
in the rest of the paragraph."
```

After turning it into a corpus, tokenizing (punctuation removed, lowercased) and n-gramming (1, 2 and 3 n-grams), we get to the following tables:

```
# table1Gram:
           ngram frequency
 1:            a        15
 2:           of         8
 3:    paragraph         8
 4:           in         6
 5:           is         6
---
72:      because         1
73:           it         1
74:     controls         1
75:      happens         1
76:         rest         1

# table2Gram:
             ngram frequency
  1:   a paragraph         7
  2:  paragraph is         4
  3:          is a         3
  4:      group of         3
  5:       a group         2
 ---
111:        in the         1
112:      the rest         1
113:       rest of         1
114:        of the         1
115: the paragraph         1

 # table3Gram:
```

```
                  ngram frequency
   1:     a paragraph is         4
   2:     paragraph is a         2
   3:         a group of         2
   4: group of sentences         2
   5: paragraphs are the         1
  ---
 113:     happens in the         1
 114:        in the rest         1
 115:        the rest of         1
 116:        rest of the         1
 117:   of the paragraph         1
```

Now suppose we want to check the probabilities of the *final words* that succeed the *string* `a paragraph` in this corpus. Since we are talking about 3 words in total, we must find the matches in `table3Gram`:

```
                    ngram frequency
1:          a paragraph is         4
2:         a paragraph can         1
```

Both `is` and `can` can succeed `a paragraph`. Now our aim is to apply Scary Equation in order to calculate `P_KN(is|a paragraph)` and `P_KN(can|a paragraph)`: that is what the left side of the equation means.

To proceed, let's first decompose that gigantic equation into three parts I will call `firstTerm`, `lambda` and `Pcont`:

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \underbrace{\boxed{\frac{\max(c_{KN}(w_{i-n+1}^{i}) - d, 0)}{c_{KN}(w_{i-n+1}^{i-1})}}}_{\texttt{firstTerm}} + \underbrace{\boxed{\lambda(w_{i-n+1}^{i-1})}}_{\texttt{lambda}} \underbrace{\boxed{P_{KN}(w_i|w_{i-n+2}^{i-1})}}_{\texttt{Pcont}}$$

Let's first turn aaaaaaall our attention to `firstTerm`. Things start quite rough, as that `c_KN` thingy is dependent on context. Here is its definition (from the source material again, eq. 4.36):

$$c_{KN}(\cdot) = \begin{cases} \text{count}(\cdot) & \text{for the highest order} \\ \text{continuationcount}(\cdot) & \text{for lower orders} \end{cases}$$

This will be the Count Equation.

Since we are at the moment dealing with the highest order n-gram, we will address only this case by now. Being so, the numerator in `firstTerm` means 1) the frequency (= count) of that full n-gram (= `c_KN`) minus a discount factor called *d,* **or** 2) zero, whatever is largest (`max`). According to the awesome material here and here, *d* is equal to 0 at the highest order n-gram. The denominator is the frequency of the *string* (the full n-gram devoid of the *final word*), which in this example is equal to the frequency of `a paragraph *`: the frequency of `a paragraph is` plus the frequency of `a paragraph can`. Therefore, for word `is`, `firstTerm(is) = 4/(4+1) = 0.8`, and for word `can`, `firstTerm(can) = 1/(4+1) = 0.2`.

Now, `lambda` has a function of its own (as in here again, eq. 4.34):

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})}|\{w : c(w_{i-1}, w) > 0\}|$$

Remember that in the highest order n-gram case, `d = 0`, then `lambda = 0`, but just for illustration, let's examine the other components of the equation. According to the source material, the denominator in the fraction is the frequency of the *semifinal word* in the special case of a 2-gram, but in the recursion scheme we are developing, we should consider the whole *string* preceding the *final word* (well, for the 2-gram case, the *semifinal word* **is** the whole *string*). The term to the right of the fraction means the number (not the frequency) of different *final word types* succeeding the *string*; since in our case *final words* `is` and `can` succeed the *string* `a paragraph`, this number is 2. Notice that `lambda` is a function of the *string,* not of the *final words,* therefore it is the same for all *final words*. Therefore, here `lambda = 0/(1+4)*2 = 0`.

Then we get to `Pcont`, which stands for the *continuation probability* (I didn't come up with that acronym from nothing). Referring again to the source material (eq. 4.32):

$$P_{\text{CONTINUATION}}(w_i) = \frac{|\{w_{i-1} : C(w_{i-1}w_i) > 0\}|}{\sum_{w'_i}|\{w'_{i-1} : c(w'_{i-1}w'_i) > 0\}|}$$

The numerator means the number of *different string types* preceding the *final word,* and the denominator means the number of *different possible n-gram types* (or, simply put,

the length of the n-gram table we are considering at the time, which in this case is `table3Gram`).

The word `is` can be preceded by 3 different string types:

```
                ngram frequency
1:     a paragraph is         4
2:         a paper is         1
3: among sentences is         1
```

The word `can` is preceded by only 1 string type:

```
             ngram frequency
1: a paragraph can         1
```

Since `table3Gram` has 117 entries, `Pcont(is) = 3/117 = 0.026` and `Pcont(can) = 1/117 = 0.009`.

Pulling it all together:

```
P(is|a paragraph) = 0.8 + 0 * 0.026 = 0.8
P(can|a paragraph) = 0.2 + 0 * 0.009 = 0.2
```

If we were to predict the next word after the string `a paragraph`, in this context, we would say it should be `is`, because its probability is highest (and equal to 80%).

Nice right? But you might (or rather should!) be asking, where is the recursion part? What is it for? Didn't you forget something?

Well, if we cannot find a match in the highest order n-gram table (for example, there is no match in this corpus for `the paragraph`), then we drop down to the next lower order table, discarding the first word in the *string*. So we should search for `paragraph` starting an n-gram in `table2Gram`, which returns:

```
             ngram frequency
1:   paragraph is         4
2: paragraphs are         1
3:  paragraphs in         1
4:  paragraph can         1
```

Then we repeat the whole process, but with two caveats.

One: Now we are considering the second condition in that Count Equation from `firstTerm`. In this case, `c_KN` is equal to the *continuation count*. The *continuation count* is defined as the number of different *word types* preceding the *string* we are considering at the time. In the case of the numerator in `firstTerm`, we are talking about the continuation count for the *full n-gram* (`i-n+1` stands for the *first word*, and `i` stands for the *last word*; the way they are displayed in reference to `w` means *all words from first to final*). In the case of the denominator, that is simply the *string* (= *all swords from first to second final*). Since we are counting *words* preceding, at least, the *full string* (in the denominator; the numerator stands for the *full string* + the *final word* = the *full n-gram*), it just makes sense counting them in the immediately higher order n-gram table (because nothing precedes the *string* in the current order). From the example, let's examine only the most frequent 2-gram, `paragraph is`.

For the numerator, we should check how many *words types* precede `paragraph is`:

```
             ngram frequency
1: a paragraph is         4
```

Only 1 word type, namely `a`. Then, `c_KN(paragraph is) = 1`.

For the denominator, we check how many *word types* precede *paragraph:*

```
                     ngram frequency
1:      length a paragraph         1
2:   sentences a paragraph         1
3: constitutes a paragraph         1
4:   paragraph a paragraph         1
5:          is a paragraph         1
6:      styles a paragraph         1
```

```
7:  ultimately a paragraph         1
8:        of the paragraph         1
```

There are 8 word types, then `c_KN(paragraph) = 8`.

Two: Since we are not at the highest order n-gram anymore, we must set *d* to 0.75 (as recommended in the video previously mentioned).

Therefore, `firstTerm(is|paragraph) = max(1-0.75, 0)/8 = 0.25/8 = 0.03125`.

The rest of the process is the same, the same! `lambda` will now be different than 0, and the final probability will be slightly corrected upwards.

One last comment, or more like a curiosity: If we cannot find a corresponding match to our query and drop down all the way to `table1Gram`, then we have to consider that the *string* preceding the *final word* is **empty**, and that will make it equal for all n-grams in that table. If one succeeds in implementing a loop with the rules mentioned above, though, the loop will take care of this observation on its own.

Arh-wooooooooo!

NLP Kneser Ney Smoothing Numerical Example