- Comment your code
- sep file to explain
- code should work
- dicts, dicts wihtin dicts

Linguistique de corpus en anglaise
Semestre 1 – 2021/2022

- An algorithm to write reviews
-

Project Semester 1
Estimating a Language Model to Generate Wine Reviews

1. *Estimating the Language Model*

The objective of this exercise is to build a program capable of automatically writing wine reviews, like the ones found on the Wine Spectator website, using a language model.

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.

This information is at the heart of speech recognition or machine translation systems and it can also be used to generate sentences similar to those produced by a human.

The probabilities (parameters) of the language model will be estimated from the reviews that can be found on the Wine Spectator website. These reviews can be found in the project folder on github (https://github.com/armandstrickernlp/CL_class_inalco).

The file text_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 wine reviews.

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 ?

b. Write the function *make_trigrams* which returns the list of successive triplets from a string of words.
For example, when the string “I love chocolate ice-cream.” is passed to this function, it should return the list :
[("I", "love", "chocolate"), ("love", "chocolate", "ice-cream"), ("chocolate", "ice-cream", ".")].

c. Write a function *make_conditional_probas* that estimates all of the probabilities p(c|a,b) from a file passed as argument. One way of doing this is by first constructing a count table which looks like the following :

dict of tuple: dict of occurences

```
{('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}),
```

sum of all occurences : 1+1 for ('like', 'chocolate')

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.

occurences of a,b, c ->

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

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}})
```

d. Two words (BEGIN NOW) have been added to the beginning of each review. Why ? Why are 2 words added and not 1 or 5 for example ?

As soon as you see END , stop calculating probability for that line.

Once I have BEGIN NOW, I'm going to look at the proba of words that can follow.
Probability distribution

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 two words w_{i-2}, w_{i-1} chosen during steps $i-1$ *and* $i-2$. The history is then updated $h \longleftarrow w_{i-1}, w_i$ and the procedure is repeated until the token END is generated.

h = history

begin history with
(BEGIN, NOW)
update with (NOW, word)
and so on

If a probability distribution X that generates events a_i with probability p_i is represented by a dictionary whose keys are the a_i and values the probabilities p_i, it is possible to generate an element a_i with its associated probability p_i using the following function:

use this code , to pick next word?

```
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 X, this function will generate *ice-cream* 50% of the time and *pudding* the remaining 50%.

techer's sol: send (BEGIN NOW)

my_sol- to initialise history, make a list of the sorted the occurrence list by highest items and pass the corresponding value of the most occuring key (bi_gram)

1. How should you initialize the history ?

2. Implement this algorithm by creating [illegible]kes as argument the probability table. What do you think of the sentences obtained ?

3. Can you think of a/some way(s) to estimate the quality of the sentences produced ?