Spell Correction Report

1. Preparation

1.1 Corpus

```
from nltk.corpus import reuters,brown
```

1.2 Confusion Matrix v0.0 Calculation

Refering to the slides of lecture 3.2, I downloaded `spell-error.txt` from: Peter Norvig's list of errors, through which I calculate 4 confusion matrixes -- insertion, deletion, substitution and transposition confusion matrix. They are in the directory `confusion matrix`. The format is dict though the extention is '.txt'.

1.2.1 Python Lib -- Difflib Usage

I use a python lib named `difflib` to get differences between two words. This helps me to get the editing type and error letters of the two words. For example, the correct word is `across`, and the correspoding error is `acress`, then the output sequence of code `difflib.Differ().compare("across", "acress")` is `acr- o+ess`.

From the distance between the positions of '+' and '-' , I can get the edit type of these two words. The table below lists part of situations. The details are in the `confusion_matrix.py`. Besides, there are more complex situations if the edit distance is 2 between the pair.

condition	edit type	example	diff
abs_dis=2	sub	"across", "acress"	acr-o+ess
abs_dis=3 && diff[add +1] == diff[del+1]	trans	"caress", "acress"	-ca+cress
other situations	insertion	"cress", "acress"	+acress
other situations	deletion	"actress", "acress"	ac-tess

1.2.2 Pre-Processing

Because "-" will be in the result of the comparison between two words, the "-" in the error_dict (generated according to `spell-error.txt`) is ambiguous. If '-' appears in the pair, I used '#' to replace it. This bug took me a lot of time to fix.

1.2.3 Matrix Meaning

Here I give some examples of the item in the 4 matrixs.

- Insertion matrix: {'ab': 4}, this item means that the error "a typed as ab" appears 4 times in the `spell-error.txt` file.
- Deletion matrix: {'ab': 10}, this item means that the error "ab typed as a" appears 10 times in the `spell-error.txt` file.
- Subsitution matrix: {'ab': 1}, this item means that the error "ab typed as ba" appears once in the `spell-error.txt` file.
- Transposition matrix: {'ab': 3}, this item means that the error "a typed as b" appears 3 times in the `spell-error.txt` file.

1.2.4 Calculation Function

```
def cal_matrix(error_dict)
```

1.3 Confusion Matrix v1.0 Calculation

From count_1edit.txt I downloaded the file `count_1edit.txt` , according to which I got 4 other matrixes, whose formats vary from the confusion matrixed I mentioned below. These 4 matrixes perform better than Confusion Matrix v0.0

1.3.1 Calculation Function

```
def cal_matrix_v2(path='count_1edit.txt')
```

1.4 Gram_dict Calculation

We can count the frequencies of unigram, bigram and trigram from the corpus loaded previously. Then we can get a gram_dict, which will be used in the language model. The function is in the file `ngram.py`.

1.5 Candidates Generation

Once a spelling mistake appears, we have to generate its corresponding candidates.

For a spelling error in a sentence, I first used four editing operations -- insertion, deletion, substitution and transposition to generate the mis-spelt word's candidates. For every candidate, c, in the candidates list, it costs a lot to go through the whole vocab file to find if c is the probable correct candidate. For plenty of sentences within mis-spelt words, this method is not feasible. (The functions are still in the script -- `get_candidates.py`).

So I transfered to another more efficient method. Then I used trie to generate candidates, the theory of which is prefix matching. For example, the trie of the word "apple" is like this: `{'a': {'p': {'p': {'l': {'e': {'/': {}}}}}}}`. The char '/' marks the end of a word.

1.5.1 Function and Example

```
def get_candidates_from_trie(vocab,word, edit_distance=1)
...
>> print(get_candidates_from_trie(vocab,"acress"))
output: ['caress', 'access', 'actress', 'across', 'acres', 'acres']
```

2. Noisy Channel Model

The formulas of the channel model of 4 editing types are as follows. I have given 3 types of channel models in terms of different smoothing methods or confusion matrixes.

$$P(x|w) = \begin{cases} \frac{del[w_{i-1},w_i]}{count[w_{i-1}w_i]} & \text{if deletion} \\ \frac{ins[w_{i-1},x_i]}{count[w_{i-1}]} & \text{if insertion} \\ \frac{sub[x_i,w_i]}{count[w_i]} & \text{if substitution} \\ \frac{trans[w_i,w_{i+1}]}{count[w_iw_{i+1}]} & \text{if transposition} \end{cases}$$

2.1 Edit Type Function

Edit type decides which matrix we are to use in the channel model. So we have to calculate the edit type between the error and the candidate. Besides, we should get two error letters as the key to search corresponding matrix. Here I also used `difflib`.

```
def get_operation_letter(word, candidate):
...
  return edit_type, edit_distance, x, y
```

2.2 Example

Meaning: The operation between the pair is transposition. And the corresponding letters are 'c' and 'a', meaning that one types 'c' as 'a'. The editing distance equals 1.

```
>> print(get_operation_letter(word="acress", "caress"))

output: ([0, 0, 0, 1], 1, {'ins': '', 'del': '', 'sub': '', 'trans': 'c'},
{'ins': '', 'del': '', 'sub': '', 'trans': 'a'})
```

3. Language Model

Considering adding operation is faster than multiplying and to avoid underflow, I do **logarithmic computation** to the probability result.

3.1 Unigram

The general definition of the 1-gram language model is like this:

$$P(w_1, w_2, \ldots, w_m) = \prod_{i=1}^{m} P(w_i)$$

However, it's unnecessary to calculate the whole sentence's probability. Because except for the position of the misspelled word, the other words are the same. It can be simplized that we only need to calculate the frequency of each candidate, using a gram_dict. Then we pick the max one.

$$w_i = \underset{w_i}{argmax}\ \frac{c(w_i)}{V}$$

3.2 Bigram

$$P(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

To fully consider the context, I calculate probability of the candidate with its two neighbors.

$$P(w_i|w_{i+1}) = \frac{c(w_i, w_{i+1})}{c(w_{i+1})}$$
$$P(w_i|w_{i-1}, w_{i+1}) = P(w_i|w_{i+1}) * P(w_i|w_{i-1})$$
$$log(P(w_i|w_{i-1}, w_{i+1}) = log(P(w_i|w_{i+1})) + log(P(w_i|w_{i-1})))$$

3.3 Trigram

$$P(w_1, w_2 \dots w_m) = \prod_{i=1}^{m} P(w_i|w_{i-2}w_{i-1})$$

3.4 Smoothing Methods

The model performed poor if not using any smoothing method. So I have tried the following smoothing methods.

3.4.1 Add-1 Smoothing

The formula is as following. Here $|V|$ is the size of the corpus.

$$P_{Add-1}(w_i|w_{i-n+1} \dots w_{i-1}) = \frac{C(w_{i-n+1} \dots w_i) + 1}{C(w_{i-n+1} \dots w_{i-1}) + |V|}$$

3.4.2 Add-k Smoothing

Actually Add-1 smoothing is a special case when k is assigned as 1. ($0 < k < 1$)

$$P_{Add-k}(w_i|w_{i-n+1}\dots w_{i-1}) = \frac{C(w_{i-n+1}\dots w_i) + k}{C(w_{i-n+1}\dots w_{i-1}) + k|V|}$$

Take Bigram as example:

$$P_{Add-k}(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i) + k}{c(w_{i-1}) + k * V}$$

3.4.3 Kneser Ney Smoothing

$$P_{continuation}(w) = \frac{|w_{i-1} : c(w_{i-1}, w) > 0|}{\Sigma_{w'}|w'_{i-1} : c(w'_{i-1}, w') > 0|}$$

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

$$P_{KN}(w_i|w_{i-1}) = \frac{max(c(w_{i-1}, w_i) - d, 0)}{c(w_{i-1})} + \lambda(w_{i-1})P_{continuation}(w_i)$$

$$P_{KN} = log(P_{KN}(w_i|w_{i-1})) + log(P_{KN}(w_i|w_{i+1}))$$

As I mentioned in the Bigram part, I calculate probability of the candidate with its two neighbors.

4. Experiments

Corpus	Language Model	Smoothing method	Counfusion matrix	Accuracy
brown	Unigram	No smoothing	v0.0	69.0%
reuters	Unigram	Add-1	v0.0	73.0%
reuters	Unigram	Add-k, k = 0.01	v0.0	75.2%
reuters	Unigram	Add-k, k = 0.01	v1.0	81.9%
brown	Bigram	Add-k, k = 0.01	v1.0	80.5%
reuters	Bigram	Add-1	v1.0	85.6%
reuters	Bigram	Add-k, k = 0.01	v1.0	89.7%
reuters	Bigram	Kneser Ney	v1.0	**92.1%**

5. Some Tips

- Put the process of vocab loading, corpus loading, trie getting, and gram_dict calculation at first. Do not put them in seperate functions, or the time compexity will be extremely bad.
- When getting the gram_dict, the object `Counter` offered by `Python collections` increases the time complexity dramatically. Calculating by going through the whole corpus performs better than it.
- For the language model, at first I calculated the probability of each whole sentence, whose error word is replaced by the candidate, which worsens the time complexity. Then I turned to calculate the probability of the gram using smoothing methods. The class and the function is still in the file `ngram.py`, which helps us to calculate the probability of a sentence. If interested, run the code below.

```
sentence = "I love you"
cate = reuters.categories()
V,corpus= get_corpus(cate=cate)
ng = ngram1(n=1, sent=sentence,corpus=corpus, V=V)
print(ng.cal_sent_prob())
```

- Smoothing method matters !!

6. References

1. difflib
2. How to build a trie
3. Trie树,加快单词查找效率
4. 自然语言处理之数据平滑方法
5. 自然语言处理中N-Gram模型的Smoothing算法