Spell Correction - PJ report

By Shihan Ran - 15307130424

I. Environment

- Python 3.6.1
- Package:
    - numpy 1.13.1 and nltk 3.2.3

II. Theory

1. Expression

We use probabilities to choose the most likely spelling correction for word w.
That means, when we are trying to find the correction word c, out of all possible candidate corrections, we can maximizes the probability that c is the intended correction, given the original word w:

$$argmax_{c \in candidates} P(c|w)$$

And by Bayes' Theorem we know that:

$$argmax_{c \in candidates} P(c) \frac{P(w|c)}{P(w)}$$

Since $P(w)$ is same for every candidate c, the question is equivalent to:

$$argmax_{c \in candidates} P(c)P(w|c)$$

2. Models

- **Selection Mechanism:** *argmax*
  We choose the candidate with the **highest** probability.
- **Language Model:** $P(c)$
  The probability that c appears as a word of English text.
- **Candidate Model:** $c \in candidates$
  This tells us which candidate corrections, c, to consider.
- **Channel Model:** $P(w|c)$
  The probability that w would be typed in a text when the author meant c.

III. Data

1. Training Corpus

```
from nltk.corpus import reuters
corpus_raw_text = reuters.sents(categories=reuters.categories())
```

Actually, I've tried `reuters` and `brown-news`, and I found that reuters is more suitable for this project.
The Accuracy of reuters is higher about **8%**.

2. Test data

1,000 senteces created by TA in `./testdata.txt`

IV. Project Structure

Selection Mechanism:

Simply use `max()`.

Language Model

A model that computes either of these:

$$P(W) = P(w_1, w_2, \dots, w_n),\ P(w_n|w_1, w_2, \dots, w_{n-1})$$

is called a language model.
We use **Chain Rule** and **Markov Assumption** to compute $P(W)$.

Candidate Model

We use **Edit Distance** to find which candidate corrections, c, to consider.

Channel Model

At first, I take it that all known words of **edit distance 1** are infinitely more probable than known words of **edit distance 2**.
Then we **don't need** to multiply by a $P(w|c)$ factor, because every candidate will have the same probability.
Later, TA gave us some inspiration that using `spell-error.txt` to generate **confusion matrix.**

Talk is cheap, show me the code!

`def preprocessing(ngram):`

- read the vocabulary from `vocab.txt` and store it to a list
- read testdata from `testdata.txt` and preprocessing it
- preprocessing the corpus and **generate the count-file of n-gram**

`def language_model(gram_count, V, data, ngram):`

- given a sentence or phrase or word, predict the probability
- do everything **in log space** to void underflow (also adding is faster than multiplying).
- I've tried using add-λ smoothing and simple backoff

```
# unigram
Tanin/V = -13.3879685383
```

```
Taiwan/V = -8.56768697273
Darwin/V = -13.3879685383
Twain/V = -13.3879685383

# bigram
first quarter/first = -7.41537424018
first quartet/first = -13.3879685383

# trigram
for the while/for the=-13.3879685383
for the whole/for the=-11.1922992368
for the whoe/for the=-13.3879685383
```

`def make_trie(vocab):`

- turn the vocabulary_list into a trie
- this change of data structure will **improve the code effectiveness** from 15mins to **30s**

`def get_candidate(trie, word, path='', edit_distance=1):`

- it will return the candidate list of the error word according to the given edit_distance

```
>>> get_candidate(trie, 'miney', path='', edit_distance=1)
>>> ['money', 'mined', 'miner', 'mines' ,'mine']

>>> get_candidate(trie, 'wsohe', path='', edit_distance=2)
>>> ['she', 'shoe', 'some', 'sore', 'sole', 'soe', 'swore', 'whole', 'whore', 'whose',
'whoe', 'wrote', 'whoe', 'wove', 'woke', 'wore', 'woe', 'wohd']
```

`def edit_type(candidate, word):`

- Method to calculate edit type for single edit errors.

```
could coul ('Deletion', 'd', '', 'c', 'cd')
mine miney ('Insertion', '', 'y', '#y', '#')
barely barels ('Substitution', 'y', 's', 's', 'y')
revenues ervenues ('Reversal', 'er', 're', 're', 'er')
```

`def load_confusion_matrix():`

- Method to load Confusion Matrix from external data file.

`def channel_model(x,y, edit, corpus):`

- Method to calculate channel model probability for errors.

```
could coul -1.60943791243
soul coul -9.48322601519

three trhee -5.35185813348
tree trhee -12.7105501626
```

`def spell_correct(vocab, testdata, gram_count, corpus, V, trie, ngram, lamd):`

- get the candidate_list and find the one has the higest probability using **language model and channel model**
- correct the error word in testdata
- write it to result

```
1 1 protectionst protectionist
2 1 Tkyo Tokyo
3 1 retaiation retaliation
4 1 tases taxes
5 1 busines business
```

V. Evaluation

Fix ngram at unigram, change corpus:

Corpus	Accuracy
Reuters	85.80%
Brown - news	81.10%

I've tried some **other categories** in reuters, but the accuracy doesn't increase apprarently, only within 2%.

Fix Corpus at Reuters, use channel model:

n-gram	p(w|c) = 1	compute p(w|c)
unigram	85.80%	89.70%

Using `spell-error.txt` to generate **confusion matrix** and then use matrices to compute **p(w|c)**.

Fix Corpus at Reuters, change smooth:

n-gram	Add-1	Add-λ
unigram	85.40%	85.40%
bigram	87.60%	**93.10%**
trigram	84.90%	87s.20%

> The reason why the accuracy of unigram is higher than bigram and trigram may be that the smoothing is not good for language modeling, because the number of zeros isn't so huge.

Best Accuracy: 93.10%

VI. Some thoughts:

Except coding, I've spent much time on :

- Using better **data structure** can improve the effectiveness greatly (from 15mins to 25s or even better)
    - use **dict**, **trie**, **.data**, **counter**
- I found **bugs in a blog**, and I email the writer to discuss with him, helping him improve his code
- Try to make my code more **pythonic**

VII. Reference

1. 动态规划求编辑距离
2. 让你的Python代码更加pythonic
3. 鹅厂面试题：英语单词拼写检查算法？
4. NLP 笔记 - 平滑方法(Smoothing)小结
5. norvig: spell-errors.txt
6. How to Write a Spelling Corrector
7. Damn Cool Algorithms, Part 1: BK-Trees